1.0

1.1

1.25

4.5
5.0
3.6
3.3
2.7
2.5

2.8
3.2
3.6
4.0

2.5
2.2
2.0
1.8

1.4

1.6

MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

DTIC
S D
JAN 1 3 1984
H
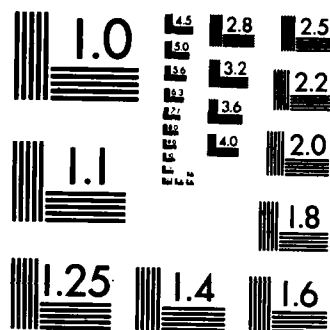
A USER-ORIENTED MICROPROCESSOR SHELL
COMMAND LANGUAGE INTERPRETER

by

Dennis J. Ritaldato

and

David J. Smania

September 1983

Thesis Advisor: Ronald Modes

Approved for public release; distribution unlimited

84 01

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
	AD-A136 809	

4. TITLE (and Subtitle)	5. TYPE OF REPORT & PERIOD COVERED
A User-Oriented Microprocessor Shell Command Language Interpreter	Master's Thesis September, 1983
	6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)	8. CONTRACT OR GRANT NUMBER(s)
Dennis J. Ritaldato and David J. Smania	

9. PERFORMING ORGANIZATION NAME AND ADDRESS	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
Naval Postgraduate School Monterey, California 93943	

11. CONTROLLING OFFICE NAME AND ADDRESS	12. REPORT DATE
Naval Postgraduate School Monterey, California 93943	September, 1983
	13. NUMBER OF PAGES 116

14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report)
	UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Command Language, Shell, Interpreter, User-friendly

Accession For

NTIS GRA&I	☑
DTIC TAB	☐
Unannounced	☐
Justification	

By

Distribution/

Availability Codes

| Dist | Avail and/or Special |

A-1

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The design of a microprocessor command language, RSCL, is discussed. RSCL provides the capability of building variable shell environments on a standard microprocessor system. These environments present a menu driven, screen oriented user interface as opposed to the line oriented interface of current operating systems.
The RSCL is a straightforward, easily understandable and complete computer programming language. Designed according to specific command language guidelines, it allows the user to make maximum utility of his skills. ──> (Continued)

DD FORM 1473 EDITION OF 1 NOV 68 IS OBSOLETE
1 JAN 73

S/N 0102-LF-014-6601

1 SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

ABSTRACT (Continued)   Block # 20

A prototype implementation and sample program runs are included.
These illustrate the design features and serve as a test platform
for future research.

A User-Oriented
Microprocessor Shell Command Language Interpreter

by

Dennis J. Ritaldato
B.S.E.E. Villanova University 1974
M.S.E.E. Drexel University 1981

and

David J. Smania
Lieutenant Commander, United States Navy
B.S. Weber State College 1972
M.A. Pepperdine University 1979

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 1993

Authors: _____

_____

Approved by: _____

Thesis Advisor

_____
Daniel Davis
Second Reader

_____
Uno R Kodres
Chairman, Department of Computer Science

_____
K. T. Marshall
Dean of Information and Policy Sciences

3

ABSTRACT

The design of a microprocessor command language, RSCL, is discussed. RSCL provides the capability of building variable shell environments on a standard microprocessor system. These environments present a menu driven, screen oriented user interface as opposed to the line oriented interface of current operating systems.

The RSCL is a straightforward, easily understandable and complete computer programming language. Designed according to specific command language guidelines, it allows the user to make maximum utility of his skills.

A prototype implementation and sample program runs are included. These illustrate the design features and serve as a test platform for future research.

ACKNOWLEDGMENTS

TRADEMARKS

CP/M is a registered trademark of Digital Research. Display Manager is a trademark of Digital Research.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

A. BACKGROUND

In the early days of computing it was simply man against the
primitive operations of the computer. There was no need for
any Command language because programming was done bit by bit
without complex interfacing. Computer systems consisted of
many tubes, a few cable connections, and possibly a periph-
eral device to display the results (output). The programmers
of the early days were considered jack of all trades. They
designed the rudimentary programs, entered them bit by bit
by re-arranging the cable configurations and should problems
arise they were the only trained maintenance technicians.
This idyllic situation did not persist for long.

Advances in computer technology especially in regards to
resources available made it imperitive that the user be
given some access mechanism to these resources. The first
system to provide such a means was the IBM 360. The system
required precise instructions to execute the system func-
tions. Unfortunately, these instructions were not self
generated like today's systems but required external media
intervention. This external media was in the form of punch
cards each containing a precise coded instruction which was
then feed into the system along with the program card deck.
The system designers either misconceived the effect of these
cards on programmers or miscalculated their abilities to
achieve an automated system. The result was catastrophic.
The first of the Command languages was a piece meal language
conceived in part as an after thought to a poor system
design. The IBM language called a JCL (Job Control Language)
did just that, it controlled the program execution by the

10

insertion of instruction cards throughout the program to manage the system's resources. The language was ambiguous, inconsistant, machine dependent and designed with little concern for the user. The impact of the IBM JCL language spawned numerous reserch efforts several of which are outlined.

During the late 1960's and early 1970's several organizations established working groups to study the JCL and OSCL (Operaing System Control Language) interface problem. The first organization to study the problem was the American Standards Institute Committee on Programming Languges (ANSI) in June of 1967 [Ref. 1]. They conducted extensive surveys of nine existing O/S systems and their control languages. Their findings concluded with a list of five recommendations

```
1. The need for a standard OSCL exists and its
   attainment is possible
2. Several features now present in O/S should
   not be included in the standard
3. None of the existing OSCL's surveyed are
   suitable as candidates for a standard language
4. There should only be a single standard OSCL
5. Piecemeal standardization should be avoided
```

Figure 1.1 ANSI OSCL Study Recommendations.

for a design proposal, figure 1.1

The Dutch established committees in September of 1971 and conducted numerous meetings under the auspicies of the Netherlands Centre for Informatics. They focused on the basic functions of job control as related to data processing and job control inputs. The committee developed a list of basic job control functions, Figure 1.2, is a synopsis of their classification of related O/S funcions.

11

```
1. Allocation (of resources)
2. Conditional Selection (of part of a job)
3. Execution (of a computer program)
4. Declaration (of job attributes)
```

Figure 1.2   Dutch JCL Committee's Basic Job Functions.

In late 1972 the CODASYL (Conference on Data System
Languages) organization conducted follow on studies to the
ANSI research. They determined that the ANSI committee had
only addressed the feasability aspect of a standardized
language and so set out to design a standard OSCL language.
Three working goals were established to guide the research:
investigate the functional requirements for communications
between the user, the functional program and the hardware;
determine the functions necessary to define a standard OSCL
language and what problems such a language would have on an
O/S; develop linguistic elements which posses these func-
tions and define a machine-independent OSCL.

Since these early studies Other organizations i.e. US
Federal Information Processing Standards (FIPS), IEEE, ACM,
British Computer Society, US Department of Defense (DOD),
etc. both governmant and privately sponsored have contrib-
uted to the research and development of several prototype
OSCL languages.

The problem of standardizing Command Languages has
perpetuated itself over the years. To date only a few
languages (systems) merit any consideration as possible
solutions.

B. PURPOSE

The purpose of this project was to design a system which will enable the user to easily define a screen oriented environment (shell) for interfacing to microprocessor based computer systems.

The shell provides an abstract view of the computer system to the user. Through it command access can be controlled and a standard JCL can be created which will operate on multiple computers and operating systems. In this way, any computer system can be tailored to perform exactly as desired for each command. In addition, the same commands can be made to execute in exactly the same manner regardless of the resident operating system. This can have substantial cost saving effects in locations where multiple computers are used. Personnel will not have to be trained for each system since all systems will operate with the same JCL.

C. SCOPE

Chapter two discusses the issues involved in the design of a command language. Guidelines for the design are also presented. The features of the command language are described in Chapter three. Chapter four discusses the factors which were involved in the design. The assumptions made, the criteria established and the decisions based on them are listed. A prototype implementation is described in Chapter five. The operation of the system from the point of view of a user creating a shell environment with the command language is discussed in Chapter six. Our conclusions and recommendations including the results of the prototype implementation are presented in Chapter seven. Appendices A, B and C include the RSCL grammar, a User's Manual for the RSCL and a CLI program source code listing of the prototype implementation, respectively.

13

II. COMMAND LANGUAGE ISSUES

A. DESIGN ISSUES

Four design issues confront the designers of any interactive Command language [Ref. 2]. First, how many modes of operation should the user be forced to learn. Second, the selection sequence of commands should be consistant and not change with varying machine implementation schemes. Third, an abort mechanism must be provided to the user to terminate a command sequence without losing the current scope or environment. Finally, a clear and concise error message system must be provided to quickly resolve syntactic and semantic problems. These design issues are not all inclusive and further issues will be brought forward as the need arises.

1. COMMUNICATION STYLES

Many CL (command language) communication styles are available today. Direct keyboard entry, using pre-defined commands, allows the user to directly control the machine operations, but requires the user to learn a new, possibly criptic, language for each OS/machine used. Another method uses keyboard response dialogue to screen prompts. This method is easier to use, but requires modification of the prompts whenever a change in functions is made. Function keys are a third method for users to communicate with the system. They are very fast and simple to use. The drawback with this method is some machines do not provide a function key option or an easy means to redefine the existing key functions. The last communication style to be mentioned is the screen menu format. This style is seen as the way of the future. Commands and data are displayed on the screen

14

in menu form. The user references the command/data by positioning the cursor at the desired field or by marking the position with a light pen. Data changed on the screen are correspondingly changed in the data base. Criptic one-line commands to the O.S. are no longer required.

Some systems (Xerox Smalltalk) provide a controlled pointer (mouse) to indicate which function is to be invoked. The Apple Lisa system uses the position of the cursor to highlight a chosen function. In either case the system is screen oriented providing the user with a simple control mechanism without the need to learn another language.

2. DESIGN GUIDELINES

Several scholars have suggested guidelines for developing command languages. Rather then repeat their offerings we have consolidated our perceptions of the primary guidelines.

- The system must be consistant. It must present the same environment to the user regardless of the basic system it is operating on.

- The system must provide the user with a command sequence which is easy to use and learn, especially the most frequently used commands.

- The system must be portable. Other machines must be able to adapt to it with minimal modification.

- The system must provide a suitable error handling process, both in presenting error messages and in saving environments.

- The system should be user interactive and provide the user with the option of selecting the level of prompt help he desires. Screen oriented displays are very helpful in selecting operations, but require complex interface buffering.

15

. Control structures should be affluent enough to
  allow the user total control of the programming
  environment.

3. **USER PROGRAMMING LEVELS**

Different levels of user motivation and programming experi-
ence must be considered when designing a multi-purpose
Command Language system. Figure 2.1 shows a rough categor-
ization of potential users into four general programming
levels.
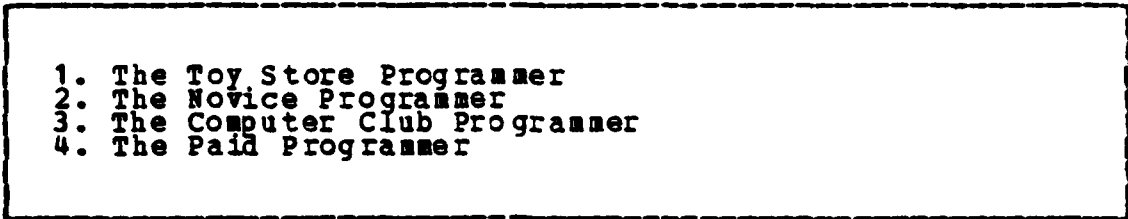
```
+-------------------------------------------------------+
|                                                       |
|   1. The Toy Store Programmer                         |
|   2. The Novice Programmer                            |
|   3. The Computer Club Programmer                     |
|   4. The Paid Programmer                              |
|                                                       |
+-------------------------------------------------------+
```

Figure 2.1   Four User Programming Levels.

The first level is the "toy store" programmer. He does not
really want to write an application program, but just wants
to know enough language tools to run a simple game program.
In general, he is in total awe of computers and makes
minimal use of their actual processing capabilites.

Progressing to the second level, the first addres-
sable command language level, we have the user who may have
attended a programming course and who is now challenged to
write a few simple application programs. The user at this
level is enthusiastic and eager to try out his new skills.
A friendly command language will motivate him to the next
level. A poorly designed command language will be frus-
trating and quite possibly curtail future computer queries.

The third level is characterized by a quantum jump
in user motivation. and usually programming skills. These

16

users really want to know how the internal system works and are willing to expend energy and their own time to learn varying system hardware and software configurations.

The final level is a grouping of two user factions into one entity. They are colloguially termed the learned computer scholars and the commercial programmers. They may perceive issues from different perspectives, yet their motives and knowledge of computer linguistics are compatible. Both require the full system resource capabilities at their immediate disposal in order to perform to their full potential.

Realistically, the majority of today's users and those who are of concern to a command language designer, fall within the final two catagories. However, care should be taken so as not to preclude use by someone at the second level.

It is easily understood why Command Languages are so universally divergent. Designing a command language to satisfy the dynamic needs of the fourth level users while still maintaining the simplicity for the novice users is not a trivial task.

4. DISPLAY FORMATS

Another issue which is receiving a great deal of attention as the state-of-the-art is the display format. Whether to display data as individual line oriented character strings or as a menu driven system. The traditional theme, driven by the hardware limitations of the past, is TTY (teletypwritter) format. i.e. Presenting a line at a time. The user responds in a similiar manner by entering data in line oriented fashion. Innovations in hardware have enabled designers to break from tradition and display whole screen prompts instantaneously.

17

The impact of these innovations has been seen is such systems as the Xerox Smalltalk and the Apple Lisa. i. e. Incorporating the traditional command language line editing commands into onscreen menu controls. The respose from critics to these nontraditonal systems has been overwhelmingly positive.

The real significance of these systems is their prime objective. They strive to provide the user with a friendly interface devoid of complex, ambiguous and inconsistant command language structures. To the "real" programmers these systems appear as a threat to their mythical man over machine syndrome. Many feel that programming is an art and a science and that these systems take away their creativity by restricting how they can address the computer. They prefer to deal direct rather than through the middleman. In reality, a friendlier interface places no such restrictions. It simply makes it more understandable so that more users can address the computer direcly. Until we overcome the system friendliness problem only those in the "real" programmers category will be willing and able to fully utilize the computer.

These systems still have some drawbacks such as overall cost and high memory requirements. Yet, given the history of the microprocessor, hardware designers will overcome the obstacles and make these features available to the average user.

The command languages of tomorrow will employ the ease of onscreen control with the user friendliness of multi-screen display.

III. R&S COMMAND LANGUAGE FEATURES

A. RSCL COMMANDS

The RSCL commands were chosen based on the guidelines outlined in chapter two. Simplicity of use, coupled with conciseness in definition and execution were paramount in choosing the RSCL commands.

Research and practical experience indicates that many command languages are either to baroque or to simple for their intended purpose. While the RSCL does not encompass all possible programming language capabilities, it does fulfill the minimum requirements of a complete language. And, it does establish a user friendly framework, a design goal stated in chapter one.

Standard command naming conventions i.e. if-then-else, put, case etc. were adopted whenever possible. The syntax structure does not deviate from established norms, while the semantics of the command language avoids ambiguity. Exception processing and type conversion is not performed nor is it implied in any of the commands. If the system does not know what you intended, it tells you via an error message. If data types do not match across an assignment operation, the assignment is not permitted.

1. BUILT IN FEATURES

Several features have been built into the RSCL which simplify both the language itself and the programs written in the language.

. Dynamic data typing (offsets declaration requirements).

. Both interactive and file processing capabilities.

. Automatic file opening and closing operations performed

19

on both the PUT and the GET commands.

. Format free statement entry.

. Statements may be entered in either upper or lower case.

2. LANGUAGE LIMITATIONS

The language, as designed, has the following limitations:

. Arrays and data structures are not defined.

. Only decimal integers may be represented.

. Floating point arithmetic is not supported.

. Unary operators are not supported.

. Exponentiation is not supported.

In the prototype implementation the following additional contraints were placed on the system:

. Loop statements may not be nested.

. Only one string variable may exist at any time.

3. LANGUAGE COMMANDS

The ten RSCL commands were chosen as the minimum number required to facilitate the requirements of a screen oriented command language.

The following is a brief description of each command. For a detailed description see the R&S Command Language User's Manual, Appendix B.

a. LET command

The LET command serves as the assignment statement. The variable on the left hand side (LHS) of the "=", receives a value from the right hand side (RHS). The RHS can be either an expression, an integer, a string, or another variable (containing a value of the same data type). If the RHS is a legal arithemetic expression, its value is computed and the

result is assigned to the variable on the LHS. Otherwise, the value of the RHS is directly assigned to the variable on the LHS.

b. The PUT command

The PUT command consists of three parts; the device, an optional line skip parameter and the data list. It outputs newlines and the items specified in the data list to the named device.

Valid devices are: "LST", the system line printer; "CRT", the user's console screen and <FN> the name of a disk file. The device name may be followed by the word "skip" (performed only once per-command). Each occurrance causes a newline character to be output. The data list contains any combination of variables and strings (a string consists of any characters contained within double quote symbols " "). The value of the variable and the actual character string, minus the quote marks, will be output.

c. The GET Command

The GET command reads data from either the user's console or from files stored on the user's disk. The device name ("CRT", <FN>) preceeds the receiving variables and indicates which medium the user wishes to access for his data.

d. IF command

The IF command is used to logically select whether or not to execute a particular set of statements. It has three components; a logical expression, a THEN set of statements and an optional ELSE set of statements. The value of the logical expression is computed. If the expression result is true, (value not equal 0), the THEN group of statements is executed. Otherwise, the ELSE group of statements is executed. If no ELSE group is included execution continues after the end of the IF statement.

21

e. The CASE Command

The CASE command provides for the execution of one or more statements contained within at least one sub_case. The sub_case is entered if it's corresponding case label matches the value of the CASE statement parameter (variable or integer). If no sub_case label matches the label of the case value, the OTHERWISE set of statements is executed.

f. The LOOP command

The LOOP command consists of two parts; the loop iteration parameter and the body of statements. All of the statements included within the body of the loop are repeated a number of times equal to the value of the loop iteration parameter. If this value is less then or equal to 0, no statements are executed.

g. The COMMENT Command

The COMMENT command performs no actual processing. Its purpose is to allow the user to document his program and to structure it in a logically understandable form. A comment begins with a ";" and, as all other RSCL statements, it terminates with a ";". Everything contained within these two semicolons is ignored.

h. The LOCATE Command

The LOCATE Command is used to determine the current location of the cursor. It returns the row and column number.

i. The POSITION Command

The POSITION Command is used to place the cursor at a particular point (row and column position) on the screen.

j. The CREATE Command

The CREATE Command is used to generate a screen template.
It consists of two parts; the template identifier and from 1
to 24 line definitions.

The template identifier is a variable name used
to differentiate one template from another. The line defi-
nitions specify up to 80 fields per line and their associ-
ated attributes.

k. The DISPLAY Command

The DISPLAY Command causes a screen template and its associ-
ated data to be output to the user's console screen. It
consists of two parts; the template identifier and an
optional set of parameters.

The template identifier is a variable name
supplied by the CREATE command when it generated the temp-
late. The parameters include a line number, a field number
and text. The line and field numbers specify exactly where
on the template the text has changed. These parameters are
returned by the display manager whenever the data in a
particular field has changed.

IV. SYSTEM DESIGN

A. DESIGN ASSUMPTIONS

Four major design assumptions were made early in the design phase. First, the integrated system is intended to operate on either 8 or 16 bit microcomputers.

Second, the interfaces between the host operating system, the command language and the Display Manager are all transparent to the user. The use of abstract interfaces between these three modules enables the system to be readily transportable to various microcomputers and operating systems.

Third, memory utilization was not considered a prime concern. The current trends in the state of the art towards larger, cheaper memories lead us to believe that the difference of one or two thousand bytes out of possibly one megabyte of storage is insignificant.

Fourth, processing speed was considered important, although not paramount in the design. Since the system is to be in continuous operation serving as the interface between the user and the imbedded operating system, some overhead is acceptable in exchange for the added capabilities. This overhead should occur during the user's "think" time rather than during actual processing.

B. DESIGN CRITERIA

Several criteria were considered during the design phase. Clarity, simplicity, portability, maintenance and upward compatibility were all key factors in designing the system. The ultimate goal was to design a system that incorporated the features outlined in Chapter two in a clear and concise

24

manner without overburdening the user. The limited number
of language commands is a direct attempt to demonstrate that
a command language can be simple and can function clearly
without an excessive number of nebulous commands. The
sample programs listed in the users manual demonstrate the
clarity of command usage.

The use of a high level system programming language,
"C", serves to grant the desired portability. "C" compilers
are available on many micro, mini and mainframe systems.
Cross-compilers should be available for those systems which
do not have a "C" compiler.

The VAX computer was used for development of the proto-
type system. Its processing capabilities and myriad of
supporting functions along with its multiprogramming oper-
ating system and the availablity of a competent support
staff made it more suitable for development than a single
user micro system. The use of any features unique to the
VAX is purely accidental. To assure program portability,
only standard "C" programming features were used.
Extensions and system dependant features must be avoided in
any implementation.

Program maintenance is supported by the use of a higher
order programming language, functional decompostion,
abstract interfaces and structured programming techniques.
The utility of these factors was directly observed during
the debug and test phases of the prototype implementation.

In addition, the use of a higher order language, the
simplicity of the language design and the avoidance of
nonstatndard features ensures some degree of upward system
compatibility.

25

C. DESIGN DECISIONS

Three major design decisions were faced during the development of the command language. First, which language should be used to implement the system. Second, certain grammar, structure and implementation conventions had to be adopted in order to ensure system integrity. Third, the interfaces between the operating system, the Command Language module and the Display Manager module was uncertain.

"C" was chosen to implement the system because of its inherant system design features. It was originally designed as a system development tool. As such, it was felt to be the most suitable language for our purposes.

In designing the RSCL grammar, standard conventions for representing the lexical ordering and syntax of the language were devised. These conventions were documented and are included with the grammar itself in Appendix A. The use of these standards was necessary in order to assure that we both interpreted the grammar in a like way and that separate modules, which were coded independently, would operate in a like manner.

Prior to initiating the actual coding phase, several sessions were held to establish programming guidelines and intermodule interfaces. Global variables, data types, error handling, system diagnostic and integration standards were defined. Any changes or variations from these established guidelines were discussed and agreed upon before being incorporated into the respective functions. This practice proved to be invaluable during the integration and test of the prototype system. No significant interfacing problems were encountered.

The last major design decision concerned interfacing the command language interpreter (CLI) with the resident operating system and with the Display Manager program. Neither of these interfaces was built into the current system.

Instead, abstract interfaces were planned for each of
these. The operating system interface will be a function
call with a character string parameter. For example, to
change the name of a file, a rename function would exist.
This function would require two parameters; the old file
name and the new file name. The file names are expected to
be complete. Information such as the disk drive designator
should be included in the name whether or not the user
enters it. The rename function would then cause the oper-
ating system to change the name of the file in whatever ways
it feels is optimal. It is transparent and irrelevant to
the (CLI), whether a separate command to the operating
system is generated or the data is sent to the BIOS or the
disk file directory is changed. In this way, a change in
the underlying operating system will require a change in
only these interface functions. It makes no difference to
the majority of the system whether a file name is changed by
an "mv" command as in VAX UNIX, an "ren" command as in CP/M,
an "r" command as in VMS, etc. The implementation of these
functions is currently the topic of a separate thesis at the
Naval Postgraduate School.

The interface to the Display Manager module was not
implemented because it was planned to use a pre-existing
program. A commercial product, called "Display Manager" is
available from Digital Research. This program does all that
we needed in the RSCL system. It also is capable of inter-
facing directly with a program written in the "C" language.
Rather than devote time to development of a new program with
similar capability, it was decided to purchase and use the
Digital Research Display Manager. Our efforts were spent
defining the display data which were of concern. This
information is included in the language grammar within the
"create" command. The actual commands telling the Display
Manager what to display are included in the grammar within

27

the "display" command.   Coding of these functions was post-
poned until the program could be purchased.   At that time
the actual interfacing parameters required can be determined
and the functions can be written.

V. IMPLEMENTATION

A. DATA ORGANIZATION

The language data organization is broken up into two parts,
local and global variables. Local variables are used within
each function to handle internal data transactions. Data
shared between two functions is passed globally. The global
variables are declared in a central file, "global.interp",
to maintain tight control over their assignment and use.
Using global variables to transfer external data decreased
the system execution time, while making the functions
cleaner and easier to integrate. The design specifications
clearly define each global variable, its use and what func-
tions utilize it's values.

Comments are generously dispersed throughout each func-
tion to aid the user in understanding its purpose. A header
is appended to the beginning of each function, listing the
other functions called and the global variables and
constants used within the function.

B. PROGRAM ORGANIZATION

Functional decomposition was used extensively throughout the
program. Three separate modules comprise the fully inte-
grated system. The O/S module is a set of functions which
defines the interface to the host's operating system. These
functions translate commands from the CLI to the native
language of the O/S. Those commands that are not native to
the host will be software emulated, if possible.

The language interpreter module has a dual function. It
interfaces with both the O/S and display modules. Programs
generated either interactively or by batch mode are

29

processed through the language interpreter. Output from the interpreter is a series of instructions to one of the other two modules.

The last link in the triad is the display module. Like the O/S module, it receives its data from the interpreter through the interface commands. The display module takes data stored in a file and transposes it onto one of the formats generated by the create command.

C. RUNTIME ERROR CHECKING

A run time error handler is built into the system and is activated when an invalid command sequence is encountered by the interpreter. All error messages are contained in a single error function. When an error is detected the error handler is called and prints a diagnostic message and amplifying information. Depending on the severity of the error either the program is terminated or control is returned to the calling function.

VI. SYSTEM OPERATION

The RSCL is capable of operating either in an interactive or a batch mode. For batch mode operation, programs can be written using any standard text editing program. The file containing the source code must be called "RSCL".

At execution time, the CLI determines if a file exists with the name "RSCL". If one is found, it is assumed to contain the source statements. The CLI then executes in a batch mode taking its input from the source file. Otherwise, the CLI reads its instructions from the user's console.

When operating in an interactive mode, the user must still follow the complete format of the language structure. That is, the program begins with the word "program", terminates with the word "end." and each statement terminates with a ";".

VII. CONCLUSIONS AND RECOMMENDATIONS

The command language design and a prototype implementation,
have been completed. This design is now reviewed to deter-
mine which of our original goals have been met or can be met
with further work and which, if any, were not found to be
feasible.

A. GOALS

The goal of this work was to design a command language which
runs on microprocessor based computer systems. The purpose
of the language was to allow rapid definition of a screen
oriented user interface. The language was to be simple,
easy to use and readily understandable. Maintainability and
portability across different machines and operating systems
were prime concerns. Processing efficiency was considered,
but only secondarily to the other factors.

B. PROBLEM AREAS

The RSCL is complete and workable as designed. Known
problem areas which are stated as constraints in the
language are not inherent problems. They can be eliminated
during future enhancements. The only area which we see as a
potential problem to the system design is related to the
Display Manager interface. The design in this area was
purposefully made generic via the principles of abstract
interfaces and information hiding. However, if the func-
tions required by the CLI are not available, some redesign
may be necessary. Because of our research in the area
before creating the design, we do not feel this is a major
concern. The CLI module should be easily interfaced to the
other two major system modules when they become available.

32

C. FUTURE WORK

In order to create a complete and deliverable product further work is required in four areas; the operating system interface routines must be completed, enhancements must be added to eliminate the constraints discussed in Chapter three, the three main system modules must be integrated, studies should be performed to determine user needs and reactions.

The operating system interface routines are already being developed under a separate thesis effort at the Naval Postgraduate School. Assuming that a copy of the Display Manager Program is obtained from Digital Research and that a "C" compiler is available for the NPS microprocessor system, the system enhancements and module integration can be accomplished under another thesis. Concurently with the system integration, research should be performed to determine sample presentation formats. They could then be created in the RSCL.

APPENDIX A
COMMAND LANGUAGE GRAMMAR

The convention used for describing the grammar of the command language is described in table I.

```
+----------------------------------------------------------------+
|                                                                |
|                          TABLE I                               |
|                                                                |
|                     Grammar Convention                         |
|                                                                |
|   SYMBOL MEANING                                               |
|                                                                |
|   < >     Used as delimiters for metasymbols                  |
|           in the grammar.  Anything contained                  |
|           within these brackets is defined                     |
|           later in the grammar.                                |
|                                                                |
|   [ ]     Used as delimiters surrounding                       |
|           optional entries.                                    |
|                                                                |
|   ' '     Used as delimiters surrounding                       |
|           literal expressions in the grammar.                  |
|           Anything within these symbols must                   |
|           appear exactly as shown.                             |
|                                                                |
|   |       Used as a logical OR.                                |
|                                                                |
|   ::=     Interpreted as "Defined as".                         |
|                                                                |
|   ( )     Used to group expressions.                           |
|                                                                |
|   **N     Used to designate a repetitive group.               |
|           Where "N" is the number of repetitions.              |
|                                                                |
+----------------------------------------------------------------+
```

Using this convention the Command Language Grammar is defined as follows:

PROGRAM ::= 'PROGRAM' <IDENTIFIER> <STATEMENTS> 'END.'

STATEMENTS ::= (<LET_STATEMENT> | <IF_STATEMENT>
                | <PUT_STATEMENT> | <GET_STATEMENT>
                | <LOOP_STATEMENT> | <CASE_STATEMENT>
                | <COMMENT> | <DISPLAY> | <CREATE>) ';'
                [<STATEMENTS>]

```
LET_STATEMENT ::= 'LET' <IDENTIFIER> '=' ( <EXPRESSION>
                  | <IDENTIFIER> | <NUMBER> | <STRING> )

IF_STATEMENT ::= 'IF' <LOG_EXP> 'THEN' <STATEMENTS>
                 ['ELSE' <STATEMENTS> ] 'ENDIF'

PUT_STATEMENT ::= 'PUT' <PUT_DEVICE> [ 'SKIP' ] <LIST>

PUT_DEVICE    ::= 'CRT' | 'LST' | <FNAME>

LIST          ::= ( <IDENTIFIER> | <STRING> ) [<LIST>]

GET_STATEMENT ::= 'GET' <GET_DEVICE> <ID_LIST>

GET_DEVICE    ::= 'CRT' | <FNAME>

ID_LIST       ::= <IDENTIFIER> [ <ID_LIST> ]

FNAME         ::= [ <CHARACTER> ':' ] <IDENTIFIER>
                  [ '.' <IDENTIFIER> ]

LOOP_STATEMENT ::= 'LOOP' ( <IDENTIFIER> | <NUMBER> )
                   <STATEMENTS> 'ENDLOOP'

CASE_STATEMENT ::= 'CASE' <IDENTIFIER> ':' <CASE_NUM>
                   'OTHERWISE:' <STATEMENTS> 'ENDCASE'

CASE_NUM   ::= (<NUMBER> | <IDENTIFIER>) ':' <STATEMENTS>
               [<CASE_NUM>]

COMMENT    ::= ';' <ANYTHING>

IDENTIFIER ::= <CHARACTER> [<SUB_ID>]

SUB_ID     ::= ['_'] (<CHARACTER> | <DIGIT>) [<SUB_ID>]

EXPRESSION ::= '(' <TERM> [ <ARITH_OPR> <SUB_EXP> ] ')'

SUB_EXP    ::= <TERM> [ <ARITH_OPR> <SUB_EXP>]

TERM       ::= <EXPRESSION> | <IDENTIFIER> | <NUMBER>

LOG_EXP    ::= '(' <LOG_TERM> [ <LOG_OPR> <SUB_LOG> ] ')'

SUB_LOG    ::= <LOG_TERM> [ <LOG_OPR> ( <SUB_LOG>
               | <LOG_EXP> ) ]

LOG_TERM   ::= <LOG_EXP> | <IDENTIFIER> | <NUMBER>

SPACES     ::= ' ' | ' ' <SPACES>

NUMBER     ::= <DIGIT> | <DIGIT> <NUMBER>

CHARACTER  ::= 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G'
             | 'H' | 'I' | 'J' | 'K' | 'L' | 'M' | 'N'
             | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U'
             | 'V' | 'W' | 'X' | 'Y' | 'Z' | 'a' | 'b'
             | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i'
             | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p'
             | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w'
             | 'x' | 'y' | 'z'

DIGIT ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7'
        | '8' | '9'

ARITH_OPR ::= '+' | '-' | '*' | '/'

LOG_OPR ::= 'EQ' | 'LT' | 'GT' | 'NE' | 'LE' | 'GE'
```

35

```
LOG_FUNC ::= 'AND' | 'OR' | 'NOT' | 'CON'

STRING ::= '''' <ANYTHING> ''''

ANYTHING ::= ( <DIGIT> | <CHARACTER> | <ARITH_OPR>
               <OTHERS> [ <ANYTHING> ]

OTHERS   ::= '!' | '(' | ')' | '$' | '!' | '@' | '#'
             | '&' | '=' | '%' | '"' | '!' | '<' | '>'
             | '.' | ',' |

DISPLAY ::= 'DISPLAY' <IDENTIFIER> [<PARAMS>]

PARAMS   ::= '(' [<LINE>] ';' [<FIELD>] ','
                 [<TEXT>] ';'

LINE      ::= NUMBER

FIELD     ::= NUMBER

TEXT      ::= ANYTHING

LOCATE    ::= 'LOCATE' <ROW> <COL>

POSITION ::= 'POSIT' <ROW> <COL>

ROW       ::= NUMBER

COL       ::= NUMBER

CREATE    ::= 'CREATE' <IDENTIFIER> <DEF_LINE>**24 'END'

DEF_LINE ::= 'DEF_LINE' <NUMBER> [ '-' <NUMBER> ]
             ( <DEF_FIELD>**80 | 'BLANK' ) 'ENDLINE' ';'

DEF_FIELD ::= 'DEF_FIELD' <IDENTIFIER> <ATTRIBUTES> ';'

ATTRIBUTES ::= '(' [<LENGTH>] ',' [<VALUE>] ','
                   [<ACCESS>] ',' [<BACKGROUND>] ','
                   [<FOREGROUND>] ',' [<VIDEO>] ','
                   [<UNDERLINE>] ',' [<INTENSITY>] ','
                   [<TYPE>] ')'

LENGTH    ::= NUMBER

VALUE     ::= ANYTHING

ACCESS    ::= 'R/O' | 'R/W'

FOREGROUND ::= DIGIT

BACKGROUND ::= DIGIT

VIDEO     ::= 'NORMAL' | 'INVERSE'

UNDERLINE ::= 'ON' | 'OFF'

INTENSITY ::= 'BRIGHT' | 'DIM'

TYPE      ::= 'ALPHANUM' | 'NUMBER' | 'CHAR' | 'STRING'
```

36

APPENDIX B
R&S COMMAND LANGUAGE USER'S MANUAL

A. INTRODUCTION

The R&S Command Language (RSCL) is designed to create micro-processor shell formats from within user designed software programs. Programs written in the language will interface with the display module to output data in the specified screen format. Menus are utilized to facilitate program entry and apprise users of available formatting options. The language uses an interpreter, written in "C", to execute the programs.

B. LEXICAL CONVENTIONS

There are seven types of tokens: identifiers, integers, strings, arithmetic operators, logical operators, logical functions and others. In general blanks, tabs, comments and newlines are ignored except as they serve to separate tokens. At least one of these characters is required to separate otherwise adjacent identifiers. The language does not incorporate any reserved words in the grammar. Each of the RSCL statements is considered a keyword when used at the beginning of a command sequence, however, since keywords are not treated as reserved words they are allowed to be assigned as identifiers latter in a command line. The semi-colon acts as a statement terminator.

1. TOKEN DESCRIPTIONS

Each word is scanned for inclusion in one of the seven identified token types. The tokens are then processed one at a time through the CLI. The following subsections describe the token formats in detail.

37

a. IDENTIFIERS

Identifiers may consist of alphanumeric characters and the underscore symbol. The first character must be alphabetic. It is optionally followed by characters, underscores or digits. Upper or lower case alphabetic characters are allowed but are not distinguished. The standard convention of not allowing an identifier to terminate with an underscore applies. Identifiers have a maximum length of ten characters and their value can be one of three types: character, string or integer.

BNF format:
```
IDENTIFIER ::= <CHARACTER>   <SUB_ID>
SUB_ID ::= '_' ( <CHARACTER> | <DIGIT> )   <SUB_ID>
```

b. NUMBERS

Numbers are formed by concatinating one or more digits onto a digit. Only digits are used to form numbers. Numbers are not re-definable data types.

BNF format:
```
NUMBER ::= <DIGIT> | <DIGIT> <NUMBER>
DIGIT ::= '0'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'
```

c. STRINGS

Strings are any ASCII print character(s) between double quotation marks (" "). The language reads "this is a string" as a single string.

BNF format:
```
STRING ::= '"'<ANYTHING>'"'
ANYTHING ::= ( <DIGIT>|<CHARACTER>|
                    <ARITH_OPR>|<OTHERS>   <ANYTHING>
```

d.   ARITHMETIC OPERATORS

Standard arithmetic operators i.e. "+", "-", "*", "/" are implemented within the language. Unary operations are not currently supported by the language.

BNF format:

```
ARITH_OPR ::= '+'|'-'|'*'|'/'
```

e.   LOGICAL OPERATORS

Alphabetic type characters i.e. "EQ", "LT", "GT", "NE", "GE", "LE" are used to perform logical operations. The first three equate to equals, less then, and greater then respectively. The last three equate to not equal, greater then or equal, and less then or equal. All expressions are required to be parenthesized i.e. (A GE B) or (4 LT 9).

BNF format:

```
LOG_OPR ::= 'EQ'|'LT'|'GT'|'NE'|'GE'|'LE'
```

f.   LOGICAL FUNCTIONS

Logical functions also use alphabetic type characters i.e. "AND", "OR", "NOT", "CON" to perform their functions. The "AND", function returns true if the two arguments bracketing the "AND" are both true. The "OR" function returns true if either of the bracketing arguments is true. The "NOT" function logically complemnts its operand. The "CON" function concatenates a string onto another string. Like the logical operators, parentheses are required in logical expressions i.e. ((Z GE T) AND (M LT W)) where Z, T, M, and W are variables or expressions which evaluate to comparible data types.

BNF format:

```
LOG_FUNC ::= 'AND'|'OR'|'NOT'|'CON'
```

g. OTHERS

The others token type is a collection of the remaining standard ASCII print character types i.e. "(", ")", "&", "%", "#", etc. These characters represent their normal meaning except where their meaning is negated i.e. "GT" replaces ">" sign in the grammar convention.

BNF format:

```
OTHERS ::= ''|'('|')'|'!'|':'|'?'|'$'|'|'|'@'|'#'|'&'|'='
           |'%'|'"'|' '|'<'|'>'|'.'|','|' '
```

C. DECLARATIONS

The language does not provide for any variable pre-declaration. New variables on the LHS or RHS, if reading in data from the CRT (screen) or a file, will be assigned the same data type as the recieved data automatically. Type conversions are not performed in the language.

D. SYNTAX

The BNF (Backus Naur Form) syntax structure for the grammar is provided in Appendix A.

E. PROGRAM STRUCTURE

All programs written in the RSCL are comprised of three parts; a header, statements to execute and a trailer. Figure B.1 shows the format of a simple program.

This sample demonstrates the overall program structure. The first line, "program sample" is the program header. Note, that it does not include a semicolon. A semicolon is a statement terminator (a semicolon is required at the end of every statement). The complete statement is "program test <executable statements> end.;".

```
program sample

  put crt skip "Enter a value for the loop count.";
  get crt a;
  loop a
    if ( a eq 2)
    then
      put crt "The value of a is " a;
    else
      get testfile.dat b;
      put crt skip "When a = 2, b = " b;
    endif;
  endloop;
end.
```

Figure B.1   Sample Command Language Program.

The second through twelfth lines are the executable state-
ments. They perform the actual processing.   The trailer is
"end.;".

    The indentation and structured appearance is optional.
The CLI ignores blanks and carriage returns.   Therefore,
multiple statements can be placed on a single line and a
single statement can be split over several lines.   Figure
B.2 shows two legal ways to write the same statements.

```
Sample of combined lines.
let a = (b+1) ; put crt a;

Sample of line splitting.
let
a= ( b
+1 )
;put crt
a;
```

Figure B.2   Example of Two Line Formatting Techniques.

41

While no one should write a program in the second
format, if the code needs to be packed, any format is accep-
table so long as variable names are not split between lines.
In that case they will be treated as two separate variables.

Now, knowing the general structure of an RSCL program,
each of the ten individual statements are discussed in the
following sections. Each statement's function and format,
the constraints on its use and the error messages which can
occur with their probable cause(s) are described.

1. The LET Statement

The LET statement is used to perform arithmetic operations
and to assign values to variables. When performing arith-
metic, the expression on the RHS must be contained within
parenthesis. The value of that expression will then be
assigned to the variable on the LHS. If no arithmetic is
required, the RHS may contain either an integer, a string or
a variable. In that case, either the integer value, the
actual string or the value of the variable will be assigned
to the variable on the LHS.

If the variable on the LHS is not defined, it is
dynamically defined according to the type and value of the
RHS. If the variable(s) on the RHS are not defined, an
error message is printed. If both variables are defined,
their types are compared to ensure that the assignment is
correct.

a. Format

A LET statement must be in the form:
    'let' <identifier> '=' (<expression> | <identifier>
                           | <number> | <string>)
Where the word "LET" is the keyword and must be the first
word in the statement. "LET" is followed by an identifier
which is treated as the LHS of the statement and will be the

recipient of the assignment. Next, the equals sign, "=", is
expected. This is used to separate the LHS from the RHS and
to show the direction of assignment. It should not be
confused with the standard relational operator "=" which
means equality. (In RSCL equality is represented by the
string "eq"). The RHS may contain only one of either an
expression, an identifier, a number or a string. Upon
execution of the let statement, the value of the RHS is
assigned to the variable on the LHS.

The expression on the RHS may be a valid arith-
matic expression containing variables and the arithmetic
operators of "+", "-", "*" and "/". These operators hold
their standard meaning of addition, subtraction, multiplica-
tion and division. The precedence of operations may be
either implied or explicitly declared by the use of paren-
theses. The implied precedence is "*" equals "/" and "+"
equals "-". While "*" and "/" are the higher precedence and
are always performed before "+" or "-". Operations of equal
precedence are performed left to right. Figure B.3 illus-
trates the LET statement format.

b. Error Types

The error messages associated with the LET statement are:
MESSAGE:

AN IDENTIFIER was expected, but ssss was found
at 1111.
EXPLANATION:

"ssss" is the token that was found prior to the
point "1111" in the input line.   Check the syntax ,of the
statement containing this line.
MESSAGE:

= was expected, but ssss was found at 1111.

43

EXPLANATION:

"ssss" is the token that was found prior to the point "llll" in the input line. An "=" was expected designating the direction of the value assignment in the let statement.

MESSAGE:

AN ARITHMETIC EXPRESSION was expected, but ssss was found at llll.

EXPLANATION:

"ssss" is the token that was found prior to the point "llll" in the input line. The RHS of the let statement did not contain a valid arithmetic expression. Most likely a matching parenthesis was omitted.

MESSAGE:

AN ARITHMETIC OPERATOR was expected, but ssss was found at llll.

EXPLANATION:

"ssss" is the token that was found prior to the point "llll" in the input line. The RHS of the LET statement did not contain a valid arithmetic operator between two identifiers.

MESSAGE:

Undefined identifier, ssss at pppp in line: llll

EXPLANATION:

"ssss" is the token that was found prior to the point "pppp" in the input line "llll". An identifier in the RHS of the let statement did not have a value. All identifiers must be set before they can be referenced.

MESSAGE:

Data type mismatch. A string type was expected at pppp in line: llll

EXPLANATION:

The LHS of the LET statement was of type string but the RHS was not.

```
let a = 7;
```

Assigns the integer value 7 to the variable
"a". If "a" is undefined it will also dynamically
declare variable "a" type "I" for integer.

```
let a = b;
```

Assigns the value cf "b" to the variable "a".
If "b" does not have a value then an error is
called. If both "a" and "b" have values then
a type check is made. Otherwise, "a" is
dynamically assigned the data type of "b".

```
let a = ((b + c) * 7);
```

Assigns to the variable "a" the resulting value of
the RHS expression. If "a" is an undefined variable
then it will dynamically receive the same data
type as the expression result or if "a" is defined
the LHS and RHS types are compared.

```
let a = "this is a test";
```

Assigns the string, "this is a test", to the
variable "a". If "a" is undefined then it will be
dynamically assigned data type "S" in the symbol
table or variable "a" is compared for data type
"S" string.

Figure B.3   SAMPLE LET STATEMENTS.

c.   Usage Constraints

A maximum of 20 operations may be nested in the arithmetic
expression.   The type checking is performed based on the
type of the right-most variable in the RHS.

2.   The GET Statement

The GET statement is used to read data into the program from
some external device and assign that data value to a program
variable.

a.   Format

A GET statement must be in the form:
  'get' <device> <list>

45

Where the word "GET" is the keyword and must be the first word in the statement. "Get" is followed by a device. This device may be either "CRT" for the user's console or the name of a file. Anything which is not "CRT" is considered a file. The name of the file may optionally be prefaced by a disk drive designator and/or suffixed by a file type. The total file name follows the file naming conventions established for the CP/M operating system.

Following the device is a list of identifiers which will receive the values as read from the input device. If the identifiers are already defined, the data will be read according to the defined type. Otherwise, the identifier's type will be set depending on the type of the data which is read. Figure B.4 illustrates the GET statement format.

```
get CRT a (b,c...);

Reads the next terminal input and assigns it to
the variable "a". If the variable "a" is undefined
it will dynamically assign the input's data type
to "a". Otherwise, it will perform a type check
on "a". More then one input can be read from the
terminal during a get statement. Type checking
is done on each receiving variable.

get <FN> a (b,c...);

Opens the designated data file <FN> and reads the
data in sequential order. The receiving variable(s)
are either dynamically assigned the input's data
type (undeclared) or a type check is performed.
```

Figure B.4   SAMPLE GET STATEMENTS.

46

b. Error Types

The error messages associated with the GET statement are:
MESSAGE:

DEVICE 'CRT' or a file name was expected, but ssss was found at llll.
EXPLANATION:

"ssss" is the token that was found prior to the point "llll" in the input line. The word 'get' must be followed by the name of the device from which to read the data.
MESSAGE:

Filename -- (<FN> [.<FT>]) -- was expected, but ssss was found at llll.
EXPLANATION:

"ssss" is the token that was found prior to the point "llll" in the input line. A file name may be up to 8 characters long and optionally prefaced by a drive designator (one character followed by a colon).

Filetype -- (<FN> [.<FT>]) -- was expected, but ssss was found at llll.
EXPLANATION:

"ssss" is the token that was found prior to the point "llll" in the input line. A file type may be up to 3 characters long and must be prefaced by a period. A file type only appears after a valid file name.
MESSAGE:

Unable to open file - check <FN> is capitalized
EXPLANATION:

The file designated in the GET statement does not exist. Check the spelling of the file name. Be sure to watch for discrepancies in capitalization.

MESSAGE:

Cannot read from the list device at pppp in line: llll

EXPLANATION:

"pppp" is the point in the input line "llll" at which the error occurred. A device type of "LST" is illegal for the GET statement.

c. Usage Constraints

There are no usage constraints in the GET statement.

3. The PUT Statement

The PUT statement is used to ouput data from a program variable to some external device.

a. Format

A PUT statement must be in the form:

'put' <device> ['skip'] <list>

Where the word "PUT" is the keyword and must be the first word in the statement. "PUT" is followed by a device. This device may be either "CRT" for the user's console, "LST" for the line printer or the name of a file. Anything which is not "CRT" or "LST" is considered a file. The name of the file may optionally be prefaced by a disk drive designator and/or suffixed by a file type. The total file name follows the file naming conventions established for the CP/M operating system.

The device is optionally followed by the word 'skip'. If included, this will cause a newline control code to be transmitted to the output device. Note, the skip is only done once per statement.

Next is a list of identifiers and/or character strings. Figure B.5 illustrates the PUT statement format.

```
put CRT a (b,c...);

Displays on the CRT (screen) the value of the
variable "a". Multiple displays (b,c...) are allowed.

put CRT SKIP (a,b...);

Skips a line prior to displaying the variable data.
The skip is performed only once prior to displaying
ing the variable(s) value(s).

put LST a (b,c...);

Toggles the printer on (providing it is turned on)
and transfers the variable(s) value(s) to it.

put LST <FN>;

Toggles the printer on (providing it is turned on)
and transfers the data contained in the designated
file <FN>.

put <FN> a (b,c...);

Opens the designated file <FN> and stores the
variable(s) value in the file. The file <FN> is
automatically closed upon statement termination.
```

Figure B.5   SAMPLE PUT STATEMENTS.

  b. Error Types

The error messages associated with the PUT statement are:
MESSAGE:

   DEVICE "CRT" or "LST" or a file name was
expected, but ssss was found at llll.
EXPLANATION:

   "ssss" is the token that was found prior to the
point "llll" in the input line. The word 'put' must be
followed by the name of the device on which the data is to
be written.
MESSAGE:

   Filename -- (<FN> [.<FT>]) -- was expected, but
ssss was found at llll.

EXPLANATION:

"ssss" is the token that was found prior to the point "llll" in the input line. A file name may be up to 8 characters long and optionally prefaced by a drive designator (one letter followed by a colon).

MESSAGE:

Filetype -- (<FN> [.<FT>]) -- was expected, but ssss was found at llll.

EXPLANATION:

"ssss" is the token that was found prior to the point "llll" in the input line. A file type may be up to 3 characters long and must be prefaced by a period. A file type only appears after a valid file name.

MESSAGE:

Undefined identifier, ssss at pppp in line: llll

EXPLANATION:

"ssss" is the token that was found prior to the point "pppp" in the input line "llll". A value must be defined for any variable before it can be output. Be sure that all designated variables are set to some value before they are referenced.

c. Usage Constraints

If the data is being put to a file, that file is opened in append mode. Therefore, if a new file is desired, the user must ensure that any previous file with that name is erased prior to executing the PUT statement.

4. The IF Statement

The IF statement executes a set of statements based on the logical value of the IF clause. If this value is true (not 0), the THEN group of statements is executed. If the IF clause value is false (0), the ELSE group of statements is executed. The ELSE group is optional. If it does not exist

50

and the IF clause value is false, the entire IF statement is ignored.

a. Format

An IF statement must be of the form:
'if' <logical expression> 'then' <statements>

Where the word "IF" is the keyword and must be the first word in the statement. "IF" is followed by a logical expression. This expression must be contained within parentheses and may be any valid combination of logical operators (eq, lt, gt, ne, le, ge), logical functions (and, or, not) variables and numbers. Precedence of operations is determined solely on the bases of parenthetical grouping.

The logical expression must be followed by the word "THEN" and the group of statements which will be executed if the logical expression is true. This group of statements terminates either with the word "ELSE" or the word "ENDIF".

If the logical expression is false, the THEN group of statements is skipped and the ELSE group is executed (if it exists). The IF statement terminates upon detection of the word "ENDIF;". Figure B.6, illustrates the IF statement format.

b. Error Types

The error messages associated with the IF statement are:
MESSAGE:
An IF statement must have a logical expression at pppp in line: llll
EXPLANATION:
"pppp" is the point in the input line "llll" at which a logical expression was expected. Check for matching parentheses.

51

```
if <logical expression> then
    <statements>
else
    <statements>
endif;

The logical expression portion is tested first. If
true, the statements in the THEN portion (any RSCL
statements) are executed in order. The statements
contained in the ELSE (optional) portion are
executed only when the IF condition returns false.
The IF statement is terminated by an ENDIF.
```

Figure B.6   SAMPLE IF STATEMENT.

MESSAGE:

THEN was expected but ssss was found at llll.

EXPLANATION:

"ssss" is the token that was found prior to the
point "pppp".   The THEN clause is mandatory in an IF state-
ment.   Be sure that all designated variables are set before
they are referenced.

MESSAGE:

ENDIF was expected but ssss was found at llll.

EXPLANATION:

"ssss" is the token that was found prior to the
point "pppp".   An IF statement must terminate with the word
"ENDIF".

   c.   Usage Constraints

There are no usage constraints for an IF statement.

   5.   The LOOP Statement

The LOOP statement repeats a set of statements a specified
number of times. Any number of repetitions may be specified
via either a number constant or a variable entry.

52

a.   Format

A LOOP statement must be in the form:

'loop' (<identifier> | <number>) <statements> 'endloop;'
Where the word "LOOP" is the keyword and must be the first
word in the statement. "LOOP" is followed by either a
number or an identifier which gives the number of times the
loop is to be executed. The loop checks this value before
execution. If the loop value is <= 0, the statements in the
loop are skipped. Otherwise, the inner statements are
repeated until the loop counter reaches 0. The loop counter
cannot be changed once the loop has begun executing. Even
if the identifer used for the loop counter is altered, the
loop will not be affected. Figure B.7 illustrates the LOOP
statement format.

```
loop a
     <statements>
endloop;

The variable "a" contains the number of
iterations that the statements contained
within the loop will be executed. Any
combination of valid RSCL statements
is allowed.

locp 7
     <statements>
endloop;

The only difference in this statement is the loop
counter is in integer form vice identifier form.
The loop execution sequence is not altered.
```

Figure B.7   SAMPLE LOOP STATEMENT.

53

b.   Error Types

The error messages associated with the LOOP statement are:
MESSAGE:

Undefined identifier, ssss at pppp in line: llll
EXPLANATION:

"ssss" is the token that was found prior to the
point "pppp" in the input line "llll". A value must be
defined for any variable before it can be used as a loop
counter.   Be sure that all designated variables are set
before they are referenced.
MESSAGE:

An integer or variable loop count was expected
but ssss was found at llll
EXPLANATION:

"ssss" is the token that was found prior to the
point "llll".   A locp counter can only be an integer or an
identifier.

c.   Usage Constraints

Nested loops cannot be used.

6.   The CASE Statement

The CASE statement   executes a set of statements based upon
the case variable.   If one of the cases matches the value of
the case variable then that set of statements is executed.
If none match,   then the OTHERWISE set of statements is
executed.

a.   Format

The CASE statement must be in the form:
  'case' <identifier> ':' case_num
                'otherwise:' <statements> 'endcase'

Where the word "CASE" is the keyword and must be the first word in the statement. "Case" is followed by an identifier and a colon. This is the case variable. Each of the cases that follow begin with either a number or an identifier followed by a colon. This value is compared with the value of the case variable. If they are equal, then all the statements in the case element (up to the next case number) are executed. If no case number matches the case variable then the statements in the otherwise clause are executed. The CASE statement is terminated with the word "ENDCASE" and a semicolon. Figure B.8 illustrates the CASE statement format.

```
case a:
   b: <statements>
   c: <statements>
   6: <statements>
   otherwise
endcase;

The case statement uses a variable or an integer
to indicate which case element will be invoked.
The "a" represents the data type of the case
element index. If none of the case elements are
invoked then the otherwise case element is
executed. Any valid RSCL statement is allowed.
```

Figure B.8   SAMPLE CASE STATEMENT.

b.   Error Types

The error messages associated with the CASE statement are:

MESSAGE:

Undefined identifier, ssss at pppp in line: llll

EXPLANATION:

"ssss" is the token that was found prior to the point "pppp" in the input line "llll". A value must be

defined for any variable before it can be used as a loop counter. Be sure that all designated varialbes are set before they are referenced.

MESSAGE:

-- : -- was expected but ssss was found at pppp.

EXPLANATION:

"ssss" is the token that was found prior to the point "pppp". A case variable must be followed by a colon.

MESSAGE:

OTHERWISE was expected but ssss was found at pppp.

EXPLANATION:

"ssss" is the token that was found prior to the point "pppp". A CASE statement must include an OTHERWISE clause to handle the event when no labled case value was matched.

c. Usage Constraints

There are no usage constraints for the case statements.

7. The CREATE Statement

The create function was not coded because the interface between the CLI and display modules is unknown. The create module was designed to interface with a commercial product. The product is still enroute to the school. When coded the create module will assign attribute values to specified fields. The resulting template is then utilized for data display through the display module.

8. The DISPLAY Statement

The display function was intended to be an external commercial product purchased from a local vendor. Unfortunately, the supply system was uncooperative and the product never arrived. As designed, display manager takes the output data

56

and transposes it ontc the requested screen shell created in
the create module.

F. GENERAL ERROR HANDLING

The system and syntax error handler messages are formatted
as follows:

("** SYNTAX or SYSTEM ERROR ****")**
                    (ERROR MESSAGE(S))

MESSAGE:

Symbol table exceeded.

EXPLANATION:

The maximum length of the symbol table was exceeded, too
many variables in the program.

MESSAGES:

Premature end of input encountered.

EXPLANATION:

The program ended without a proper terminator i.e. END.
Program could be in the middle of a command when the input
terminates.

MESSAGES:

Unrecognized character, ssss in line: llll.

EXPLANATION:

"ssss", a non ASCII type token, was encountered prior to
the point "llll" in the input line.

MESSAGES:

String length excceds (132) in line llll

EXPLANATION:

The token prior to the point "llll" in the input line
exceeds the maximum sring length of (132).

MESSAGES:

PROGRAM was expected, but ssss was found at llll.

EXPLANATION:

Program's must start with the constant "PROGRAM" followed by the program name.

MESSAGES:

AN IDENTIFIER was expected, but ssss was found at line 1111.

EXPLANATION:

This could have several meanings. LHS's of let statements require an identifier (variable). Data file reads also require a variable to receive transfered data.

MESSAGES:

END. was expected, but ssss was found at line 1111.

EXPLANATION:

An input following a statement must be either another statement or an (END.).

MESSAGES:

No legal Command Language statement was found. prior to the point 1111 in the input line.

EXPLANATIONS:

This error message is only invoked during the first statement following the program name.

MESSAGES:

expected at ssss in line 1111.

EXPLANATION:

Semicolons terminate all statements. Check the statement at the indicated line.

APPENDIX C
PROGRAM SOURCE CODE LISTING

```
/* R & S Command Language
 *
 * Last update: 22 Sep 1983
 *
 ***************************************************************
 *                  CONSTANT DEFINITIONS                  *
 */
#define debug        0
#define debugcase    0
#define debugget     0
#define debugif      0
#define debuglet     0
#define debugloop    0
#define debugput     0
#define debugstate   0
#define false        0
#define true         1
#define maxsym       25
#define devsiz       15
#define linesiz      132
#define loop_lst_siz 10
#define stringsiz    132
#define symsiz       10
#define optorsiz     20
#define oprandsiz    40
#define EOFILE       ' '
#define NEWLINE      '0
#define id_token     1
#define str_token    2
#define int_token    3
#define arith_op_token 4
#define log_op_token   5
#define log_func_token 6
#define other_token    7


/***********************************************************************
 *                  GLOBAL VARIABLE DEFINITIONS                       *
 */
  FILE *output, *input, *source, *loop_file;
  char LOOP_FILE[20], *loopptr;          /* file name for loop statements*/
  char sav_dev[devsiz];                  /* device name for put & get    */
  char put_dev[devsiz];                  /* device name for put statement*/
  char get_dev[devsiz];                  /* device name for get statement*/
  char symtype;                          /* type of symbol I or C        */
  char symid[symsiz], *idptr;            /* actual symbol char string    */
  char string[stringsiz], *sptr;         /* character string             */
  char token[symsiz], *tptr;             /* actual token char string     */
  char line[linesiz], *lptr;             /* current input line           */
  char loop_lst[loop_lst_siz][linesiz];/*statements repeated in loop*/
```

```
int loop_cnt;                          /* loop statement counter, used */
                                       /* by getline to repeat statemnt*/
int token_type;                        /* type of token               */
int symval;                            /* value of symbol             */
int numsym;                            /* number of symbols active    */
int exp_result;                        /* result of arith expressions */
int opr_value;                         /* precedence values of arith_op*/

struct {                               /* symbol table                */
  char id[symsiz], *sidptr;            /*    symbol name              */
  int value;                           /*    symbol value             */
  char type;                           /*    symbol type (I or C)     */
} symbol[maxsym], *symptr
```

```
/*
 * This is the main routine for the Command Language Interpreter.
 * It calls "statements" to process all other statements.
 * If main completes successfully the interpreter exits.
 *
 * Functions used:  error(11/12/14/51), next, statements
 * Global used: token, token_type
 * Constants used: id_token
 *
 * Author: Dennis J. Ritaldato
 * Last update: 22 Sep 1983
 */
#include <stdio.h>
#include "global.interp"
main ()
{
  LOOP_FILE[0] = "";
  strcpy(LOOP_FILE,"LOOPZZZZ");
  loop_cnt = 0;                                    /* init loop counter */
  numsym = 0;                                       /* init symbol table */
  source = fopen("RSCL","r");                  /* open source file for */
                                             /* command language program */

  next();
  if ( strcmp(token,"PROGRAM") )
    error (11);
  next();
  if ( token_type != id_token )
    error (12);
  next();
  if ( ! statements() )
    error (51);
  if ( strcmp(token,"END") )
    error (14);
  next();
  if (token[0] != '.')
    error (14);
  exit();
}
```

```c
/*
 * Statements checks the token to determine if it is a reserved word
 * indicating the beginning of a command language statement.
 * If found the corresponding functionis called to process the
 * statement. Then statement calls itself to look for more statements
 * and returns true.
 * Functions used: case_statement, create, display, error(53),
 *                  if_statement, let_statement, loop_statement, next,
 *                  put_statement get_statement,
 * Globals used: token
 * Constants used: none
 *
 * Author: Dennis J. Ritaldato
 * Last update: 19 Sep 1983
 */
#include <stdio.h>
#include "global.interp"
statements ()
{
#if debugstate
printf("Entered statements with token of %s.0,token);
#endif
  if ( !strcmp(token, "LET") )
  { next();
    let_statement();
  }
  else if ( !strcmp(token, "IF") )
  { next();
    if_statement();
  }
  else if ( !strcmp(token, "PUT") )
  { next();
    put_statement();
  }
  else if ( !strcmp(token, "GET") )
  { next();
    get_statement();
  }
  else if ( !strcmp(token, "LOOP") )
  { next();
    loop_statement();
  }
  else if ( !strcmp(token, "CASE") )
  { next();
    case_statement();
  }
  else if ( !strcmp(token, ";") )
  { next();
#if debug
printf(" COMMENT found.0);
#endif
    while ( strcmp(token, ";") )
```

62

```
        next();
    }
    else if ( !strcmp(token, "DISPLAY") )
    { next();
      display();
    }
    else if ( !strcmp(token, "CREATE") )
    { next();
      create();
    }
    else
      return(false);
    if ( token[0] != ';' )
      error(53);
    next();                                      /* bypass ;            */
    statements();
    return (true);
}
```

63

```c
/*
 * The scanner scans the input stream for tokens which are either:
 *     identifiers   - alpha:alphanum | _
 *     integers      - digit:integers
 *     strings       - anything except ;
 *     logical ops   - EQ | LT | GT | NE | LE | GE
 *     arith ops     - * | - | + | /
 *     logical funcs - AND | OR | NCT | CON
 *     others        - any other ASCII character
 *
 * Functions used: error(4), getline
 * Globals used: line, lptr, log_opr, log_func, loop_cnt, loop_list,
 *               loop_lst_cnt, loop_lst_ptr,
 *               sptr, string, token, token_type, tptr
 * Constants used: arith_op_token, id_token, int_token, symsiz,
 *                 log_op_token, other_token, str_token,
 *                 log_func_token
 * Author: Dennis Ritaldato
 * Last update: 22 Sep 1983
 */
#include <ctype.h>
#include <stdio.h>
#include "global.interp"
next ()
{
  int i = 0;

  tptr = token;
  *tptr = NULL;
  token_type = 0;
                                              /* if end of line       */
  if ( (*lptr == NULL) || (*lptr == NEWLINE) )
    getline();                                /*     get new line      */
  while ( (*lptr == ' ') || (!isascii(*lptr) ) )/* skip blanks        */

    if ( (*lptr == NULL) || (*lptr == NEWLINE) ) /* if end of line   */
      getline();                              /*     get new line      */
    else
      ++lptr;
  }
  if ( isalpha(*lptr) )                        /* is token an identifier? */
  { for ( i = 0; isalpha(*lptr)|| isdigit(*lptr) || (*lptr == '_'); )
      if ( i++ < symsiz )
        *tptr++ = upper( lptr++ );
    *tptr = NULL;
    if ( !log_opr() && !log_func() )
      token_type = id_token;
    return;
  }
  else if ( isdigit(*lptr) )                   /* is token an integer? */
  { while ( isdigit(*lptr) )
      *tptr++ = *lptr++;
```

```
       *tptr = NULL;
       token_type = int_token;
       return;
    }
  switch (*lptr) {
  case '"':                                    /* is token a char string? */
    sptr = string;
    token_type = str_token;
    ++lptr;
    for (i=0; (*lptr != '"'); ++lptr)
    { if ( *lptr == NULL )                      /* if end of line */
        getline();                                   /* get new line */
      if (i++ < stringsiz)
        *sptr++ = *lptr;
      else
        error(5);
    }
    ++lptr;                                    /* bypass second " */
    *sptr = NULL;
    return;
  case '+': case '-':                          /* is token an arithmetic op?*/
    opr_value = 1;
    token_type = arith_op_token;
    *tptr++   = *lptr++;
    *tptr     = NULL;
    return;
  case '*': case '/':
    opr_value = 2;
    token_type = arith_op_token;
    *tptr++   = *lptr++;
    *tptr     = NULL;
    return;
  case '(': case ')': case '[': case ']';/*is token another symbol?*/
  case '{': case '}': case ':': case ';';
  case '!': case '@': case '#': case '$';
  case '%': case '^': case '&': case '~';
  case '<': case '>': case ',': case '.';
  case '=': case '?': case '|': case '\';
  case '_': case '`': case '\'';
    token_type = other_token;
    *tptr++ = *lptr++;
    *tptr = NULL;
    return;
  default:
  /*error(4);*/
#if debug
printf("--Unrecognized char %c with value %d found.0,*lptr,*lptr);
#endif
    *lptr++;
    next();
    return;
  }  /* end switch */


                                65
```

```
}                                               /* end next */
```

```
/*
 *    Getline reads the next line either from the input stream
 * if the loop counter,"loop_cnt" is 0 or from the loop statement
 * list, "loop_lst" if the loop counter is greater than 0.  It
 *  decrements the loop counter each time a line is read.
 *  Each line read, regardless of source is placed into an array of
 * characters called "line".
 * If an EOFILE is encountered an error message is printed out and
 * the program terminates.
 * Otherwise, the line pointer is reset to the beginning of the
 * line and the function returns.
 *
 * Functions used: error(3)
 * Globals used: line, lptr, loop_cnt, loop_list, loop_lst_cnt,
 *                   loop_lst_ptr
 * Constants used: arith_op_token, id_token, int_token,
 *                   log_func_token
 *
 * Author: Dennis Ritaldato
 * Last update: 22 Sep 1983
 */
getline()                                       /* begin getline        */
{
  int i;

  for (i=0; i < linesiz; i++)                   /* clear line buffer    */
    line[i] = NULL;

  if (loop_cnt > 0)                             /* read from the loop list? */
  {
    if ( fgets(line,linesiz,loop_file) == EOFILE )
    { fclose(loop_file);
      if (--loop_cnt > 0 )
      { loop_file = fopen (LOOP_FILE, "r");
        fgets(line,linesiz,loop_file);
      }
    }
    lptr = line;
    return;
  }
  else if ( source != NULL )
  { if( fgets(line,linesiz,source) == EOFILE )/*read from file RSCL*/
      error(3);
#if debug
printf("--Source line read.0);
#endif
  }
  else
  { if ( gets(line) == EOFILE )                /* read from the terminal   */
      error(3);
#if debug
printf("--CRT line read.0);
```

67

```
#endif
   }
#if debug
printf("--The new line is: %s0,line);
#endif
   lptr = line;
   return;
}                                       /* end getline              */
```

```
/*
 * Upper converts a lower case ASCII character to upper case ASCII.
 * Any characters which are not lower case ASCII are ignored.
 *
 * Author: Dennis J. Ritaldato
 * Last update: 14 Sep 1983
 */
upper(c)
  char *c;
{
  if ( ('a' <= *c) && (*c <= 'z') )        /* if lowercase        */
    *c = *c + 'A' - 'a';                    /* convert to uppercase*/
  return(*c);
}
```

```c
/*
 * Log_opr examines the current token to determine if it is a
 * logical operator.
 * If so, it sets the token_type appropriately.
 *
 * Author: Dennis J. Ritaldato
 * Last update: 15 Sep 1983
 */
log_opr()                                          /* begin log_opr      */
{
  if ( strlen(token) != 2)
    return(false);
  tptr = token;
  switch(*tptr) {
  case 'E':
    if (*++tptr != 'Q')
      return(false);
    break;
  case 'N':
    if (*++tptr != 'E')
      return(false);
    break;
  case 'G': case 'L':
    if ( (*++tptr != 'T') && (*++tptr != 'E') )
      return(false);
    break;
  default:
    return(false);
  }
  token_type = log_op_token;
  return(true);
}                                                  /* end log_opr        */
```

70

```
/*
 * Log_func examines the current token to determine if it is a
 * logical function operator.
 * If so, it sets the token_type appropriately.
 *
 * Author: Dennis J. Ritaldato
 * Last update: 14 Sep 1983
 */
log_func()                                      /* begin log_func      */
{
  if ( (!strcmp(token,"AND")) || (!strcmp(token,"OR"))
     || (!strcmp(token,"NOT")) || (!strcmp(token,"CON")) )
  { token_type = log_func_token;
    return(true);
  }
  return(false);
}                                               /* end log_func        */
```

```c
/*
 * This procedure adds a new symbol to the symbol table.
 * Increments numsym
 * creates a new symbol table entry with the values contained in
 * symid, symval and symtype
 * return
 * Author: Dennis J. Ritaldato
 * Last update: 13 Sep 1983
 */
#include <stdio.h>
#include "global.interp"
addsym()
{
  int i;
  if (numsym > maxsym)
    error(2);
  symptr = &symbol[numsym++];
  for (i=0; symid[i]!=' '; ++i) {
    symptr -> id[i] = symid[i];
    symptr -> value = symval;
    symptr -> type  = symtype; }
#if debug
printf("ADDSYM entered. Numsym = %d0,numsym);
#endif
  return;
}
```

```
/*
 * This procedure assigns the value contained in symval to the
 * symbol indicated by symptr.
 * Author: Dennis J. Ritaldato
 * Last update: 13 Sep 1983
 */
#include <stdio.h>
#include "global.interp"
setvalue()
{
  symptr -> value = symval;
  return;
}
```

```
/*
 * LOOKUP searches the symbol table for a match on symid and
 * symbol.id. If found,
 *    set symptr, symval and symtype from the contents of the
 *    symbol table, return true
 * else
 *    symptr, symval and symtype remain unchanged
 *    returns false
 * Author: Dennis J. Rtialdato
 * Last update: 13 Sep 1983
 */
#include <stdio.h>
#include "global.interp"
lookup()
{
   int i;
   for (symptr = &symbol[0]; symptr <= (symbol + numsym); ++symptr)
     for (i=0; symptr->id[i] == symid[i]; ++i)
       if (symid[i] == ' ') {
          symval = symptr->value;
          symtype = symptr->type;
          return (true); }
   return (false);
}
```

```
/*
 *This function assignes values to variables. The LHS (left hand side)
 * variable must be an identifier.  The only exception is when a string
 * is assigned then the LHS variable is the global array string. RHS's
 * can be either an expression, integer or a declared identifier with
 * a value of the identifier stored in the symbol table.  Expressions.
 * of any length are accepted. Unary minus operations are not
 * supported in this version.
 *
 * Functions used: addsym,error(12/18/55/57), expression, lookup, next
 *                 setvalue
 * Globals used: exp_result, symid, symtype, symval, token, token_type
 * Constants used: id_token, int_token
 *
 * Author: David J. Smania
 * Last Update 22 Sep 83
 */
#include <stdio.h>                      /* Link standard I/C            */
#include "global.interp"                /* link all program constants   */

 char operator[optorsiz];               /* Declare let_statement variables */
 char savetype;
 int operand[oprandsiz];
 int a,b,m,n,marker,last_prec;

let_statement ()
 {                                      /* Entering let statement       */
  char savetoken[symsiz];               /* Declare local variables      */
  int sav_value,addflag;

#if debug
printf("LET_STATEMENT entered.0);
#endif

/****************** EVALUATE LEFT HAND SIDE ************************/

  if (token_type != id_token)          /* Check for identifier         */
   {
    error (12);                        /* Error token not identifier   */
    return;
   }
  else
      strcpy(savetoken,token);         /* Save token name              */
  next ();
  if (strcmp(token,"=") == 0)          /* Check for = token            */
    next ();
  else
   {
    error(18);                         /* Missing '=' operator         */
    return;
   }
```

75

```
/*********************** RHS CHECK **********************************/
/*
 * The expression function first determines if the RHS is an
 * expression.  If so, then it evaluates the expression and
 * returns the result to exp_result.  Error checking is performed
 * throughout the function.
 */

   if (expression ())                     /* Check for 2nd arg = expression */
      sav_value = exp_result;             /* Exp result saved               */
   else
     if (token_type == id_token)
       {
        strcpy(symid,token);              /* Load symid for lookup          */
        if (lookup ())
          {
             sav_value = symval;          /* Save variable value            */
             savetype = symtype;          /* Save variable type             */
          }
        else
          error (55);                     /* Variable not in symbol table   */
        next ();
       }
   else
     if (token_type == int_token)
       {
        savetype = 'I';
        sav_value = atoi(token); /* Save integer value                      */
        next ();
       }
   else
     if (token_type == str_token)
       {
        symtype = 'S';
        next ();
        strcpy(symid,savetoken);
        if (! lookup ())
          {
            symval = 0;
            addsym ();
            return;
          }
        if (symtype == 'S')
          return;
        error (57);
        return;
       }
   else
     {
        error(24);                        /* Not exp, string, id, int       */
        return;
     }
```

76

```
/************************** STRCPY CHECK *****************************/

   strcpy(symid,savetoken);          /* Load symid for lookup           */
   if (lookup ())
     if (savetype == symtype)
       {
         symval = sav_value;
         setvalue ();        /* Assign values to symbol table variables*/
       }
     else
       error (57);                    /* Variable not in symbol table     */
   else
     {
       symval = sav_value;
       symtype = savetype;
       addsym ();                      /* Add a new variable to symbol table*/
     }
   }
```

77

```
/******************** EXPRESSICN FUNCTION **************************/
/*
 * EXPRESSION determines if the token is a valid arithmetic
 * expression.  An arithmetic expression is defined as a term
 * optionally followed by a arithmetic operator and a subexpression.
 * A term is either an expression, an identifier, a number or a
 * string.  A subexpresion is a term optionally followed by an
 * operator and a subexpression.
 *    If a valid expression is found, it's value is stored in the
 * variable "exp_result" and true is returned. Ctherwise false is
 * returned.
 *
 * Functions used: error(22/50), lookup, next, pop, pushopratot,
 *                 pushidoperand, set_prec
 * Globals used: exp_result, syrid, symtype, symval, token, token_type
 * Constants used: arith_op_token, id_token, int_token
 *
 * Author: David J. Smania
 * Last Update 22 Sep 83
 */

expression ()
 {

 m = 0;
 n = 0;
 last_prec = 0;

    if ((strcmp(token,"(") == 0))        /* Check for '(' lead of exp */
     {
      pushoprator ();                    /* Push '(' on stack         */
      next ();
      if (token_type == int_token)       /* Check for integer RHS     */
        savetype = 'I';
      else
        if (token_type == id_token)      /* Check for identifier RHS  */
         {
           strcpy(symid,token);
           if (lookup ())
            savetype = symtype;
           else
            savetype = 'C';
         }

/******************** LOOP THROUGH RHS *****************************/

    while (token(0) != ';')              /* Loop until ';' is read    */
     {
       if (strcmp(token,"(") == 0)
        {
           pushoprator ();
           next();
```

78

```
              }
          else
          if (strcmp(token,")") == 0)      /* Enter pop routines       */
              {
                pop ();
                set_prec ();
                next ();
              }
          else
          if (token_type == id_token)    /* Lookup identifiers       */
              {
                strcpy(symid,token);
                if (! lookup ())
                  error (55);
                else
                    {
                      pushidoperand ();
                      next ();
                    }
              }
          else
          if (token_type == int_token) /* Push integer tokens       */
              {
                symval = atoi(token);
                pushintoperand ();
                next ();
              }
          else
                                          /* Check operator precedence */
          if (token_type == arith_op_token)
              if (check_pri ())
                  {
                    pushoprator ();
                    next ();
                  }
              else
                  {
                    pop ();
                    set_prec ();
                  }
          else
            error (21);
      }

/********************* END WHILE LOOP ****************************/

    if ((operator[0] = '(') && (operator[1] = ')'))
          {
            exp_result = operand[--m];
            symval = exp_result;
            return (true);
          }
```

```
        else
            error (22);
    }
    else
        return (false);

}
```

```
/****************** PUSH INTEGER FUNCTION *************************/
/*
 * This function pushes the incomming integer token onto the stack
 * operand[m].
 *
 * Author: David J. Smania
 * Last Updata 25 September 83
 */

   pushintoperand ()
     {

       operand[m] = atoi(token);
       ++m;
     }


/******************** PUSH IDENTIFIER  FUNCTION ******************/
/*
 * This function pushes the identifier value operands onto the
 * stack operand[m].
 *
 * Author: David J. Smania
 * Last Updata 24 September 83
 */

   pushidoperand ()
     {

       operand[m] = symval;
       ++m;
     }


/**************** PUSH OPERATOR FUNCTION ************************/
/*
 * This function pushes the incomming operator onto the stack
 * operator[n].
 *
 * Author: David J. Smania
 * Last Updata 23 September 83
 */

   pushoprator ()
     {

       operator[n] = token[0];
       ++n;
     }
```

81

```
/****************** CHECK PRIORITY FUNCTION *********************/
/*
 * Check the incoming operator precedence with the existing highest
 * precedence, last_prec, value. Modify if incomming is greater.
 *
 * Author: David J. Smanial
 * Last Update 23 September 83
 */

check_pri ()
    {

    if (opr_value > last_prec)
         {
          last_prec = opr_value;
          marker = n;
          return (true);
         }
    else
       if (opr_value == last_prec)
          return(true);
    else
       return (false);
    }
```

82

```
/************************* POP FUNCTION ****************************/
/*
 * Pop the operators and operands off their respective stacks
 * according to the token read.

 * Author: David J. Smania
 * Last Update 23 September 83
*/

pop ()
   {
     int i,done;

     done = 0;
     --n;

     if (strcmp(token,")") == 0)             /* Pop until '(' is found  */
p
        while (operator[n] != '(')
           {
             --m;
             a = operand[m];
             --m;
             b = operand[m];
             check_token ();
           }
        else
          while (n >= marker)                /* Pop until lower precedence */
          {
            --m;
            a = operand[m];
            --m;
            b = operand[m];
            check_token ();
          }
      return;
   }
```

83

```
/********************** SET PRECEDENCE ****************************/
/*
 * Set the precedence variable last_prec to the highest precedence
 * operator in the stack operator[n].
 * Author: David J. Smania
 * Last Update 23 September 83
 */

set_prec ()
 {
  int i,done;

    done = 0;
    ++n;
    operator[n] = token[0];
    for (i=0; ((i<=n) && (!done) && (operator[i] != ')')); i++)
      {
        if ((operator[i] == '+') || (operator[i] == '-'))
          {
            last_prec = 1;
            done = true;
            marker = i;
          }
        else
          last_prec = 0;
      }
    return;
 }


/********************** PERFORM ARITHMETIC *************************/
/*
 * Perform arithmetic operations based on operator[n] found. Store
 * results in operand[m].
 *
 * Author: David J. Smania
 * Last Update 23 September 83
 */

check_token ()
  {
      if (operator[n] == '+')
        {
          operand[m] = (a + b);
          ++m;
          --n;
        }
      else
        if (operator[n] == '-')
          {
            operand[m] = (b - a);
            ++m;
            --n;
```

84

```
                }
        else
           if (operator[n] == '*')
               {
                   operand[m] = (a * b);
                   ++m;
                   --n;
               }
        else
           if (operator[n] == '/')
               {
                   operand[m] = (b / a);
                   ++m;
                   --n;
               }
        return;
   }

/******************** END LET_STATEMENT ****************************/
```

```
/*
 * This procedure receives data from either the screen (CRT) or a
 * resident file. The function first checks for the user's requested
 * display device then responds to the user's data requests. Two
 * types of data input requests are available to the user: from a
 * file on the user's disk; or a variable stored in the symble table.
 * Global variable sav_dev stores the user's device request.
 *
 * Function used: next, error(20/56), device, id_list, addsym
 * Globals used: sav_dev, get_dev, string, sptr, symtype, symval,
 *               input
 * Constants used: NULL, stringsiz
 *
 * Author: David J. Smania
 * Last Update 22 Sep 83
 */

#include <stdio.h>                       /* Link standard I/O          */
#include <ctype.h>                       /* Link integer check routine */
#include "global.interp"                 /* Link all program canstants */

char savetype;                           /* Declare local variables    */
int sav_val;

get_statement ()                         /* Entering get statement     */
{
  int bad;                               /* Declare get variables      */
#if debug
printf("GET_STATEMENT entered.0);
#endif
  if (! device())                        /* Check for device token     */
   {
    error(20);                           /* Invalid device type        */
    return;
   }
  strcpy(get_dev,sav_dev);               /* Save device name in get_dev*/
  if (strcmp(get_dev,"CRT") ==0)
  { while (id_list () )                  /* Loop until id_list is empty*/
    {
      if (savetype == 'S')               /* Check saved token type     */
        gets ( string );
      else if (savetype == 'I')
       {
         scanf (" %d",&symval);
         setvalue ();
       }
      else if (savetype == 'C')
       {
         symval = getchar();
         setvalue ();
       }
      else                               /* Identifier is unknown      */
```

86

```c
    {
      gets(string);
      sptr = string;

      for (bad=false; ( (*sptr != NULL) && (*sptr != ' ')
                        && (!bad) ); ++sptr)
        if ( ! isdigit(*sptr))          /* Is input a digit        */
          bad = true;
      if (!bad)
      { symtype = 'I';
        symval = atoi(string);
      }
      else
      { if (strlen(string) == 1)
        { symtype = 'C';
          symval = token[0];
        }
        else
          symtype = 'S';
      }
    addsym();
    }
    next ();                            /* bypass input variable   */
  }                                     /* end of while loop       */
}
else
  if (strcmp(get_dev,"LST") ==0)
  { error(56);                          /* Invalid device input    */
    return(false);
  }
else
{ input = fopen(get_dev,"r");           /* Open file to read only   */
  while (id_list () )                   /* Loop until id_list is empty */
  {
    if ( savetype == 'S')
      fgets ( string,stringsiz,input );
    else if (savetype == 'I')
    { fscanf (input," %d",&symval);
      setvalue();
    }
    else if (savetype == 'C')
    { fscanf (input," %c",&symval);
      setvalue();
    }
    else                                /* identifier is unknown    */
    {
      fscanf (input," %s",string);
      sptr = string;

      for (bad=false; ( (*sptr != NULL) && (*sptr != ' ')
                        && (!bad) ); ++sptr)
        if ( ! isdigit(*sptr))     /* Is input a digit?        */
```

87

```
              bad = true;
        if (!bad)
        { symtype = 'I';
          symval = atoi(string);
        }
        else
        { if (strlen(string) == 1)
          { symtype = 'C';
            symval = token[0];
          }
          else
            symtype = 'S';
        }
      addsym();
      }
      next ();                          /* bypass input variable      */
    }                                   /* end of while loop          */
  }                                     /* end of file processing     */
#if debugget
printf("At end of GET, token = %s0,tcken);
#endif
  return;
}                                       /* end get_statement          */
```

```
/*********************** ID_LIST FUNCTION *************************/
/*
 * The id_list function checks if the input variable is already
 * declared in the symbol table. If true, it saves the data type for
 * type checking. A data type of 'U' undefined is set otherwise.
 *
 * Function used: lookup
 * Globals used: symtype, token
 * Constants used: id_token
 *
 * Author: David J. Smania
 * Last update: 22 Sep., 1983
 */

id_list ()
{
  if (token_type != id_token)
    return(false);
  strcpy(symid,token);        /* Place token in symid for lookup check*/
  if ( lookup ())
    savetype = symtype;       /* Save token type for latter comparison*/
  else
    savetype = 'U';
  return (true);
}


/**************** END GET_STATEMENT ********************************/
```

```c
/*
 * PUT_STATEMENT outputs to either the screen (CRT)
 * or the printer (LST) data stored in a variable a
 * string or a file. The function first checks for the
 * appropriate display "device" then responds to the
 * users data requests. Two types of data requests
 * are available: a variable stored in the symbol table;
 * or a string.  Global variables put_dev, symval and
 * savetype store the device name, the token value and
 * token type respectively.
 *
 * Functions used: next, error(25), list, device,
 * Globals used: output, put_dev, sav_dev, string,
 *                  token, token_type
 * Constants used: none
 *
 * Author: David J. Smania
 * Last Update 22 Sep 83
 */


#include <stdio.h>              /* Standard I/O link          */
#include "global.interp"        /* Link all program canstants */


char savetype;                  /* Declare local variables    */
int sav_val;

put_statement ()                /* Entering put case statement */
{
#if debug
printf("PUT_STATEMENT entered.0);
#endif
  if (! device())              /* Check for device token      */
   {
    error(25);                 /* Invalid device type         */
    return;
   }
  strcpy(put_dev,sav_dev);     /* Save device name            */
  if (strcmp(put_dev,"CRT") ==0)
   { if ( !strcmp(token,"SKIP") )/* Skip a line                */
    { printf("0);
      next();                  /* bypass SKIP                 */
    }
    while (list ())            /* Loop until list is terminated*/
    {
#if debugput
printf("--List returned true with token = %s0,token);
#endif
        if (savetype == 'S')   /* Checks for token type       */
          puts(string);
        else
          if (savetype == 'I')
            printf("%d ",sav_val);
```

90

```c
          else
            printf("%c ",sav_val);
          next ();
    }                                /* end while list              */
  }                                  /* end if CRT                  */
    else
      if (strcmp(put_dev,"LST") ==0)
        while (list ())
        {
          printf("toggling printer0);
          next ();
        }
    else
    {
                                     /* Open file with 'a' attribute*/
      output = fopen(put_dev,"a");
#if debugput
printf("Opening new file  %s0,put_dev);
#endif
      if ( !strcmp(token,"SKIP") )/* Skip a line in the file */
      { printf("0);
        next();                       /* bypass SKIP                 */
      }
      while (list ())                 /* Loop until list empty       */
      {
        if (savetype == 'S')      /* Checks for token type       */
          fputs(string,output);
        else
          if (savetype == 'I')
            fprintf(output,"%d ",sav_val);
          else
            fprintf(output,"%c ",sav_val);
        next ();
      }
      fclose(output);                 /* Close data file             */
    }
#if debugput
printf("At end of put, token = %s0,token);
#endif
  return;
}
```

91

```
/*********************LIST FUNCTION**************************/
/*
 * The list function checks for the output token supplied by
 * the user. The corresponding token data values are stored
 * appropriately for later comparison.
 *
 * Functions used: next, error(55), lookup
 * Globals used: string, symid, symval, symtype, token
 * Constants used: id_token, int_token, str_token
 *
 * Author: David J. Smania
 * Last Update 22 Sep 83
 */

  list ()
    {
      if (token_type == int_token)
      {
        sav_val = atoi(token);      /* Save token value        */
        savetype = 'I';             /* Save token type         */
        return (true);
      }
      if (token_type == id_token)
      {
                      /* Place token in symid for lookup check */
        strcpy(symid,token);
        if ( lookup ())
        {
                      /* Save token type for later comparrison */
          savetype = symtype;
          sav_val = symval;         /* Save variable value     */
          return (true);
        }
        error (55);                 /* Unidentified variable   */
        return (false);
      }
      if (token_type == str_token)
      {
        savetype = 'S';
        return (true);
      }
      return (false);                        /* Error no match          */
    }                                        /* end list                */

/*******************END PUT_STATEMENT****************************************/
```

92

```
/*
 * DEVICE determines if the current token is a valid
 * I/O device name.  A valid device is defined as either
 * "CRT" for the user's console, "LST" for the line printer
 * or a filename.  The file name is structured according
 * to the file naming conventions of the CPM operating
 * system.  That is, a name optionally preceeded by a one
 * character disk drive designator with a colon and
 * optionally followed by a period with a three character
 * file type.
 *     If a valid device is found, it is stored in the
 * variable "sav_dev" and true is returned.  Otherwise,
 * false is returned.
 *
 * Functions used: next, error(26/30)
 * Globals used: sav_dev, token, token_type
 * Constants used: id_token
 *
 * Author: David J. Smania
 * Last Update 22 Sep 83
 */

#include <stdio.h>
#include "global.interp"       /* Link all program canstants */

/*********************DEVICE FUNCTION*********************/

 device ()
   {
#if debug
printf("DEVICE entered.0);
#endif
    if (token_type != id_token)
      return (false);
    if ((strcmp(token,"CRT") == 0) || (strcmp(token,"LST") ==0))
      {
#if debugput
printf("--device = %s0,token);
#endif
      strcpy(sav_dev,token);      /* Save display type */
      next ();                    /* bypass CRT | LST      */
      return (true);
      }
    strcpy(sav_dev,token);
    next ();                      /* bypass fname | drive  */
    if (strcmp(token,":") ==0)
       {
       strcat(sav_dev,token);
       next ();                         /* bypass :              */
       if (token_type != id_token)
         return (false);
       strcat(sav_dev,token);
```

```
      next ();                          /* bypass fname          */
      }
  if (strcmp(token,".") ==0)
     {
       strcat(sav_dev,token);
       next ();                         /* bypass .              */
       if (token_type != id_token)
          return (false);
       strcat(sav_dev,token);
       next ();                         /* bypass ftype          */
     }
   return (true);
  }

/*************************** END DEVICE **********************/
```

```
/* IF STATEMENT executes a set of statements based on the
 * logical value of the the IF-clause.  If this value is
 * true (not 0), the THEN-group of statements is executed.
 * If the IF-clause value is false (0), the ELSE-group of
 * statements is executed.
 *    The ELSE-group is optional.  If it does not does not
 * exist, the entire IF statement is skipped.
 *
 * Functions used: next, error(16/17/27/54), statements
 * Globals used: token, token_type, symid, symval
 * Constants used: symsiz, log_op_token, log_func_token
 *
 * Author: Dennis J. Ritaldato
 * Last update: 22 Sep 1983
 */
#include <stdio.h>
#include "global.interp"
  int log_result = 0;
  int term = 0;
  int sub = 0;
#if debugif
int level = 0;
#endif
if_statement ()
{
# if debug
printf("IF_STATEMENT entered.0);
# endif
  if (! log_exp() )
  { error(54);
    return;
  }
#if debugif
printf("--log_result at level %d is %d.0,level,log_result);
#endif
  if (log_result)
  { if ( strcmp(token,"THEN") )
    { error(16);
      return;
    }
#if debugif
printf("--THEN found.0);
#endif
    next();                     /* bypass THEN              */
    statements();               /* execute then clause      */
                                /* skip else clause         */
    while ( strcmp(token,"ENDIF") ) next();
  }
  else
                                /* skip then clause         */
  { while ( strcmp(token,"ELSE") ) next();
#if debugif
```

95

END
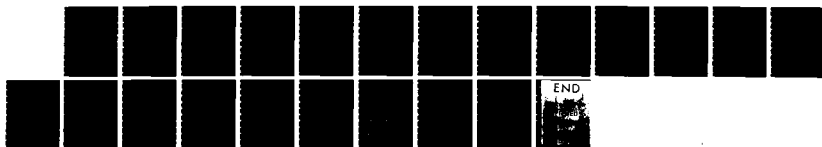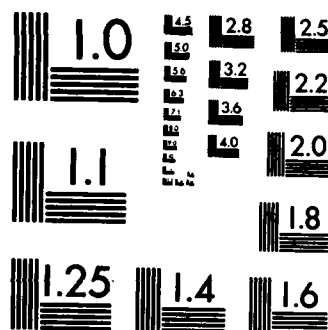
MICROCOPY RESOLUTION TEST CHART

NATIONAL BUREAU OF STANDARDS-1963-A

```
        printf("--ELSE found.0);
#endif
            next();                     /* bypass ELSE           */
            statements();               /* execute else clause   */
        }
    if ( strcmp(token,"ENDIF") )
    { error(17);
      return;
    }
    next();                             /* bypass ENDIF          */
    return;
}                                       /* end if_statement      */
```

```
/*
 * LOG_EXP determines if the current token is a logical
 * expression.  A logical expression is defined as a
 * logical term, optionally followed by both a logical
 * operator and a logical subexpression.  The entire
 * logical expression must be enclosed in parentheses.
 *     If a logical expression is found, it's value is
 * stored in the variable "log_result" and true is
 * returned.  Otherwise, false is returned.
 *
 * Author: Dennis J. Ritaldato
 * Last update: 22 Sep 1983
 */
log_exp()
{
  int lhs = 0, rhs = 0;
  char operator[symsiz];

#if debugif
printf("Entered log_exp.0);
#endif
  if (strcmp(token,"(") )
    return(false);
  next();                        /* bypass "("                    */
#if debugif
printf("----Left paren found for level %d. ",level++);
printf(" New token is %s0,token);
#endif
  if ( ! log_term() )
    return(false);
#if debugif
printf("----In log_exp, log_term returned %d ",term);
printf(" with token %s.0,token);
printf("----and token_type of %d.0,token_type);
#endif
  lhs = term;
  if ( (token_type == log_op_token)
       || (token_type == log_func_token) )
  { strcpy(operator,token);
    next();                      /* bypass operator               */
#if debugif
printf("----Logical expreession operator, %s, found.0,operator);
#endif
    if ( ! sub_log() )
      return(false);
    rhs = sub;
    log_result = compute(lhs,operator,rhs);
#if debugif
printf("----Compute returned %d.0,log_result);
#endif
    if ( !strcmp(token,")") )
    { next();                    /* bypass ")"                    */
```

97

```
#if debugif
printf("--Right paren for level %d compound expression.0,level);
printf("--With next token of %s.0,token);
#endif
        return(true);
    }
    else                    /* matching right paren not found */
    { error(27);
      return(false);
    }
  }
  else if ( !strcmp(token,")") )
  { log_result = lhs;
    next();                 /* bypass ")"                     */
    return(true);
  }
  else                      /* matching right paren not found */
    error(27);
  return(false);
}                           /* end log_exp                    */
```

```c
/*
 * SUB_LOG determines if the current token is a logical
 * subexpression.  A sub expression is defined as a
 * logical term, optionally followed by a logical operator
 * followed by either a logical subexpression or a
 * logical expression.
 *    If a logical subexpression is found, it's value is
 * stored in the variable "sub" and true is returned.
 * Otherwise, false is returned.
 */
sub_log()
{
   int lhs = 0, rhs = 0;
   char operator[symsiz];

#if debugif
printf("Entered sub_log.0);
#endif
   if ( ! log_term() )
     return(false);
   lhs = term;
   if ( (token_type != log_op_token)
     && (token_type != log_func_token) )
   { sub = lhs;
     return(true);
   }
   strcpy(operator,token);
   next();                              /* bypass operator           */
   if ( sub_log() )
   { rhs = sub;
     sub = compute(lhs,operator,rhs);
#if debugif
printf("----In sub_log, sub_log returned %d.0,sub);
#endif
     return (true);
   }
   if ( log_exp() )
   { rhs = log_result;
#if debugif
printf("----In sub_log, log_exp returned %d.0,rhs);
#endif
     sub = compute(lhs,operator,rhs);
     return (true);
   }
   return (false);
}                                       /* end sub_log               */
```

```c
/* LOG_TERM determines if the current token is a term
 * in a logical expression.  A term is defined as a
 * logical expression, an identifier or a number.
 * If a term is found, it's value is placed in the
 * global variable "term" and true is returned.
 * Otherwise, false is returned.
 */
log_term()
{
#if debugif
printf("Entered Log_term.0);
#endif
  if ( log_exp() )
   { term = log_result;
     return(true);
   }
  if (token_type == id_token)
   { strcpy(symid,token);
     lookup();
     term = symval;
     next();                              /* bypass identifier        */
     term = symval;
#if debugif
printf("----Identifier value %d was found.0,term);
#endif
     return(true);
   }
  if (token_type == int_token)
   { term = atoi(token);
     next();                              /* bypass integer           */
#if debugif
printf("----Integer value %d was found.0,term);
#endif
     return(true);
   }
  return(false);
}                                         /* end log_term             */
```

```c
/*
 * COMPUTE performs the operation specified in the
 * parameter "op" and returns a value of true or false.
 */
compute(lhs,op,rhs)
  int lhs, rhs, *op;
{
#if debugif
printf("Entered compute with %d %s %d.0,lhs,op,rhs);
#endif
  if ( !strcmp(op,"EQ") )
  { if (lhs == rhs)
      return(true);
    return(false);
  }
  if ( !strcmp(op,"LT") )
  { if (lhs < rhs)
      return(true);
    return(false);
  }
  if ( !strcmp(op,"GT") )
  { if (lhs > rhs)
      return(true);
    return(false);
  }
  if ( !strcmp(op,"NE") )
  { if (lhs != rhs)
      return(true);
    return(false);
  }
  if ( !strcmp(op,"LE") )
  { if (lhs <= rhs)
      return(true);
    return(false);
  }
  if ( !strcmp(op,"GE") )
  { if (lhs >= rhs)
      return(true);
    return(false);
  }
  if ( !strcmp(op,"AND") )
  { if (lhs & rhs)
      return(true);
    return(false);
  }
  if ( !strcmp(op,"OR") )
  { if (lhs | rhs)
      return(true);
    return(false);
  }
  if ( !strcmp(op,"NOT") )
    return( !lhs );
```

```
    if ( !stremp(op,"CON") )
    { puts("CON is not yet implementedC);
      return(false);
    }
}                                              /* end compute            */
```

```
/*
 * CASE_STATEMENT executes a set of statements based
 * upon the case variable.  If one of the cases matches
 * the value of the case variable, that set of statements
 * is executed.  If none match, the otherwise set of
 * statements is executed.
 *
 * Functions used: lookup, next, case_num,
 *                  error(23/31/32/33/55), statements
 * Globals used: token, token_type, symid, symval
 * Constants used: id_token, int_token,
 *
 * Author: Dennis J. Ritaldato
 * Last update: 22 Sep 1983
 */
#include <stdio.h>
#include "global.interp"
  int caseval;
case_statement ()
{
#if debug
  printf(" CASE_STATEMENT entered.0);
#endif
  if (token_type == id_token)
  { strcpy(symid,token);                   /* get case variable     */
    if ( ! lookup() )                      /* and save its value    */
    { error(55);                           /*    undefined variable  */
      return;
    }
    caseval = symval;
  }
  else if (token_type == int_token)
    caseval = atoi(token);
  else
  { error(23);                             /* not integer or vaiable */
    return;
  }
  next();                                  /* bypass case variable   */
  if ( strcmp(token,":") )
  { error(31);
    return;
  }
  next();                                  /* bypass :              */
  if ( ! case_num() )
  { if ( strcmp(token,"OTHERWISE") )
    { error(32);
      return;
    }
    next();                                /* bypass OTHERWISE      */
    if ( strcmp(token,":") )
    { error(31);
      return;
```

```
      }
      next();                              /* bypass :                  */
      statements();
   }
   if ( strcmp(token,"ENDCASE") )
   { error(33);
     return;
   }
   next();                                 /* bypass ENDCASE            */
}                                          /* end case_statement        */
```

```
/*
 * CASE_NUM executes a set of statements based upon
 * the case variable.  If one of the cases matches
 * the value of the case variable, that set of statements
 * is executed and true is returned.
 * Otherwise, false is returned.
 *
 * Functions used: lookup, next, case_num, error(55), statements
 * Globals used: token, token_type, symid, symval
 * Constants used: id_token, int_token,
 *
 * Author: Dennis J. Ritaldato
 * Last update: 22 Sep 1983
 */
#include <stdio.h>
case_num()
{
   int found;
   int saveval;

   for (found=false; (strcmp(token,"OTHERWISE")!=0) & (! found); )
   {
#if debugcase
printf("--Inside for loop.0);
#endif
      if (token_type == id_token)          /* maybe an identifier     */
      { strcpy(symid,token);
#if debugcase
printf("An identifier token, %s, was found.0,token);
#endif
         next();                           /* bypass identifier       */
         if ( !strcmp(token,":") )
         { next();                         /* bypass :                */
            if ( ! lookup() )
            { error(55);
              return(false);
            }
            if (caseval == symval)         /* check this case item     */
               found = true;               /* against the case value   */
         }
      }
      else if (token_type == int_token)  /* maybe an integer          */
      { saveval = atoi(token);
#if debugcase
printf("--An integer case option of %d was found.0,saveval);
#endif
         next();                           /* bypass integer          */
         if ( !strcmp(token,":") )
         { next();                         /* bypass :                */
            if (caseval == saveval)        /* check this case item     */
               found = true;               /* against the case value   */
         }
```

105

```
        }
        else                                    /* must not be the proper  */
        ( while (strcmp(token,";") )            /* case so skip rest of line*/
            next();
#if debugcase
printf("--No valid case num was found.0);
#endif
            next();                             /* bypass ;                 */
        }
    }                                           /* end-for                  */
#if debugcase
printf("--End of for loop.0);
#endif

    if ( ! found )
        return(false);
    statements();
    while ( strcmp(token,"ENDCASE") )           /* skip remaining stat(     */
        next();                                 /* in the case              */
    return(true);
}                                               /* end-case_num             */
```

```c
/* Loop repeats a set of statements a specified number
 * of times.  Any number of repetitions may be specified
 * via either a number constant or variable entry.
 *     Only one level of looping is implemented in this
 * version.  To implement multiple levels, change the
 * loop_file variable name to an array.  Then step
 * through that array.
 *
 * Functions used: next, error(23/55), lookup
 * Globals used: loop_cnt, symid, symval, loop_file
 *                token, token_type, string, sptr
 * Constants used: int_token, id_token, linesiz, NULL,
 *                 LOOP_FILE
 *
 * Author: Dennis J. Ritaldato
 * Last update: 22 Sep 1983
 */
#include <stdio.h>
#include "global.interp"
loop_statement ()
{
  int save_cnt = 0;

#if debug
printf("LOOP_STATEMENT entered.0);
#endif
  if (token_type == id_token)
  { strcpy(symid, token);
    if ( lookup() )
      save_cnt = symval;
    else
      error(55);
  }
  else if ( token_type == int_token)
    save_cnt = atoi(token);
  else
  { error(23);
    return;
  }

  next();                         /* bypass loop count variable   */

  /*                      NOTE:                          */
  /* Each line must terminate with a NEWLINE character.      */
  /* Lin_len should always point to this NEWLINE character.   */
  /* Except, when adding a string, the NEWLINE is added at the*/
  /* end of each line and at the end of the complete string.  */

  loop_file = fopen(LOOP_FILE,"w");
  while ( strcmp(token,"ENDLOOP") )
  { if ( token_type != str_token )
    { fputs(token,loop_file);       /* add identifier/number to file*/
```

107

```
          fputs("0,loop_file);
      }
     else
     { fputs("
       fputs(string,loop_file);
       fputs("
     }
     next();                        /* bypass current token        */
   }
  loop_cnt = save_cnt;
  fclose(loop_file);               /* close as write file         */
                                   /* reopen for input            */
  loop_file = fopen(LOOP_FILE,"r");

#if debugloop
printf("--In loop, bypassing token %s0,token);
#endif
  next();                          /* bypass the word "ENDLOOP"    */
#if debugloop
printf("--Leaving loop with token %s0,token);
#endif
  return;
}                                  /* end loop_statement          */
```

```c
#include <stdio.h>
#include "global.interp"
create ()
{
  printf(" CREATE entered.0);
  while (token[0] != ';')
    next();

}                                       /* end create                    */


#include <stdio.h>
#include "global.interp"
display ()
{
  printf(" DISPLAY entered.0);
  while (token[0] != ';')
    next();

}                                       /* end dislpay                   */
```

```c
/*
 * Error performs error processing.  Depending on the input
 * parameter "type" a message is printed at the user's console
 * and the function either returns or terminates the program.
 *
 * Author: Dennis J. Ritaldato & David J. Smania
 * Last update: 22 Sep 1983
 */
#include <stdio.h>
#include "global.interp"
error (type)
  int type;                              /* ERROR TYPES ARE:          */
{
  if (type <= 10)                        /* 1-10  System errors       */
  { printf ("**** SYSTEM ERROR **** 0);
    switch(type)
    { case 1:
        exit();
      case 2:
        printf ("Symbol table exceeded.0);
        exit();
      case 3:
        printf ("Premature end of input encountered.0);
        exit();
      case 4:
        printf ("Unrecognized character, %c, in line:0,*lptr);
        printf ("%s0,line );
        return;
      case 5:
        printf ("String length exceeds %d in line:0,stringsiz);
        printf ("%s0,line);
        exit();
      case 6:
        printf("Unable to open file - check <FN> is capitalized0);
        exit();
        return;
    }                                    /*       endcase            */
  }                                      /*       endif 1-10         */
  else if (type <= 50)                   /*    if 11-50 Reserved word */
  { printf ("**** SYNTAX ERROR **** 0); /* syntax errors   */
    switch(type)
    { case 11:
        printf("PROGRAM");
        break;
      case 12:
        printf("AN IDENTIFIER");
        break;
      case 14:
        printf("END.");
        break;
      case 16:
        printf("THEN");
```

110

```
          break;
      case 17:
        printf("ENDIF");
        break;
      case 19:
        printf("=");
        break;
      case 20:
        printf("Device 'CRT' or a filename");
        break;
      case 21:
        printf("AN ARITHMATIC OPERATOR");
        break;
      case 22:
        printf("AN ARITHMETIC EXPRESSION");
        break;
      case 23:
        printf("An integer or variable loop count");
        break;
      case 24:
        printf("An integer, identifier, string or expression");
        break;
      case 25:
        printf("Device 'CRT' or 'LST' or a filename");
        break;
      case 26:
        printf("Filename -- (<FN> [.<FT>]) -- ");
        break;
      case 27:
        printf("-- ) --");
        break;
      case 30:
        printf("Filetype -- (<FN> [.<FT>]) --");
        break;
      case 31:
        printf("-- : --");
        break;
      case 32:
        printf("OTHERWISE");
        break;
      case 33:
        printf("ENDCASE");
        break;
      default:
        printf ("          $$$$$ SYSTEM ERROR # 1 - %d $$$$$   ",type);
        printf (" PLEASE NOTIFY EITHER DENNIS J. RITALDATO (215) ");
        printf ("441-2107 OR DAVID J. SMANIA (408) 646-9182. 0);
        return;
    }                              /*      endcase                */
    printf (" was expected, but %s was found at %s,0,token,lptr);
    while (token[0] != ';')        /* skip remainder of line */
      next();
```

111

```
    return;
}                               /*      endif 11-30              */
else if (type <= 70)
{ switch(type)                  /*      if 51-70 General syntax  */
  { case 51:                    /*          errors              */
      printf ("No legal Command Language statement was found");
      break;
    case 53:
      printf ("; expected");
      break;
    case 54:
      printf("An IF statement must have a logical expression");
      break;
    case 55:
      printf("Undefined identifier, %s ",token);
      break;
    case 56:
      printf("Cannot read from the list device");
      break;
    case 57:
      printf("Data type mismatch.  A string type was expected");
      break;
    case 58:
      return;
    case 59:
      return;
    case 60:
      return;
    default:
      printf ("$$$$$ SYSTEM ERROR # 1 - %d $$$$$  ",type);
      printf (" PLEASE NOTIFY EITHER DENNIS J. RITALDATO (215) ");
      printf (" 441-2107 OR DAVID J. SMANIA. 0);
      return;
  }                             /*      endcase                 */
  printf (" at %s in line:112s0,letr,line);
  while ( strcmp(token,";") ) next();/* skip rest of line */
}                               /*      endif 51-70     */
}                               /* end error            */
```

LIST OF REFERENCES

1. Enslow, P. H., Command Languages, North-Holland
    Publishing Company, 1975

2. Newman, W. M. and Sproul, R. F., Principles of
    Interactive Computer Graphics McGraw-Hill, 1979

BIBLIOGRAPHY

Bidmead, C., Cifer Series 1, Practical Computing, Vol. 6, No. 4, April 1983

Ellis, J. R., A LISP Shell, Computer Science Department, Yale University, New Haven, Ct. 06520

Hendrix, J. E., Small Shell Part 2 of a North* VOS, Dr. Dobb's Journal, Vol 7, No. 1, January 1982

Irby, C., Bergsteinsson, L., Morgan, T., Newman, W., Tesler, T., A Methodology for User Interface Design, Xerox Palo Alto Research Center, 1977

Jacob, R. J. K., Using Formal Specifications in the Design of a Human-Computer Interface, Communications of the ACM, vol. 26, No. 4, April 1983

Madsen, J., CCL-A High-Level Command Language Software Practice and Experience, Vol. 9, pp. 25-30, 1979

Malcom, J. A., Brevity and Clarity in Command Languages, Sigplan Notices, vol. 16, No. 10, October 1981

Marca, D., A Repetition Construct for UNIX Version 6, Sigplan Notes, Vol. 17, No. 9, September 1982

Martin, J., Design of Man-Computer Dialogues, Prentice-Hall, 1973

Mayer, R. E., The Psychology of How Novices Learn Computer Programming ACM Computing Surveys, vol. 13, No. 1, March 1981

Miller, L. A. and Thomas, J. C. Jr., Behavioral Issues in the Use of Interactive Systems, International Journal of Man-Machine Studies, vol. 9, No. 5, September 1977

Morgan, T. P., An Applied Psychology of the User ACM ComputingSurveys, vol. 13, No. 1, March 1981

Mozelco, H., A Human/Computer Interface to Accomodate Learning Stages, Communications of the ACM, vol. 24, No. 2, February 1982

Newman, W. M. and Sproul, R. F., Principles of Interactive Computer Graphics, McGraw-Hill, 1979

Neuhold, E.J., and Weller, T., Specification and Proving of Command Programs, Acta Informatica 6, 1976

Prictchet, C.J.; Mochinacki, S. and Yang, S., RETICENT-A Command Language for Spectrophotometric Data Reduction, Astronomical Society Pacific, Vol. 94, No. 560, Aug.-Sep. 1982

Proceedings of the IFIP Working Conference on Command Languages, Command Languages, North-Holland Publishing Company, Amsterdam, 1975

Relles, N. and Price, L., *A User Interface for Online Assistance*, IEEE, 1981

Sandwall, E., *Unified Dialogue Management in the Carcusel System* Naffah, N. (editor), *Proceedings of the Second International Workshop*, North Holland, 1982

Shneiderman, B., *Human Factors Experiments in Designing Interactive Systems*, Computer, Vol. 12, No. 12, December 1979

Skjellum, A., *Expand Wildcards Under UNIX*, Dr. Dobb's Journal, Vol. 7, No. 11, November 1982

Specification and Proving of Command Programs, ACTA, Informatica 6, 1976

INITIAL DISTRIBUTION LIST

No. Copies

1. Defense Technical Information Center 2
   Cameron Station
   Alexandria, Virginia 22314

2. Library, Code 0142 2
   Naval Postgraduate School
   Monterey, California 93943

3. Department Chairman, Code 52 1
   Department of Computer Science
   Naval Postgraduate School
   Monterey, California 93943

4. Naval Air Development Center 2
   Code 501
   ATTN: Mr. Dennis J. Ritaldato
   Warminster, Pa. 18974

5. Naval Air Development Center 1
   Code 501
   Warminster, Pa. 18974

6. LCDR David J. Smania, USN 2
   27 Revere Rd.
   Monterey, Ca. 93940

7. LCDR Ronald Modes, USN, Code 52MF 2
   Department of Computer Science
   Naval Postgraduate School
   Monterey, Ca. 93943

8. Mr. Daniel Davis 1
   Department of Computer Science, Code 52
   Naval Postgraduate School
   Monterey, Ca. 93943

END

FILMED

2-84

DTIC